Title
Data-Triggered Threads /

Permalink
https://escholarship.org/uc/item/15r454pr

Author
Tseng, Hung-Wei

Publication Date
2014

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Data-Triggered Threads

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Hung-Wei Tseng

Committee in charge:

Professor Dean Tullsen, Chair
Professor Steven Swanson, Co-Chair
Professor Chung-Kuan Cheng
Professor Sorin Lerner
Professor Bill Lin

2014
The Dissertation of Hung-Wei Tseng is approved and is acceptable in quality and form for publication on microfilm and electronically:


Co-Chair

Chair

University of California, San Diego

2014
Data is not information, 
information is not knowledge, 
knowledge is not understanding, 
understanding is not wisdom.

Clifford Stoll
TABLE OF CONTENTS

Signature Page ........................................ iii
Epigraph ........................................ iv
Table of Contents ..................................... v
List of Figures ....................................... viii
List of Tables ......................................... x
Acknowledgements .................................... xi
Vita ..................................................... xiii
Abstract of the Dissertation ........................ xv
Chapter 1  Introduction .............................. 1
Chapter 2  Redundant computation ................. 6
Chapter 3  The data-triggered threads model ...... 10
  3.1  The data-triggered threads execution model ... 11
  3.2  The data-triggered threads programming model .. 12
    3.2.1  Specifying data triggers                12
    3.2.2  Composing support thread functions     14
    3.2.3  Defining skippable regions              15
    3.2.4  Code example                           16
Chapter 4  Architectural support for the DTT model ........................................ 21
  4.1  Architectural support ........................ 22
    4.1.1  ISA support                           23
    4.1.2  New hardware structures               25
  4.2  Experimental methodology .................... 27
    4.2.1  Opportunities for data-triggered threads ... 27
    4.2.2  Simulation                           29
  4.3  Results ........................................ 31
  4.4  Discussion ...................................... 36
Chapter 5  Software data-triggered threads ........ 38
  5.1  Design and implementation of software data-triggered threads .................... 39
    5.1.1  Software data structures ................. 40
    5.1.2  Detecting changes to memory contents .... 41
  5.2  Experimental methodology .................... 45
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The concept of the DTT model</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>A code example of redundant computation from gcc</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The redundant load and silent store instructions in SPEC2000 C benchmarks.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>The data-triggered thread execution model</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Two possible ways to specify triggers.</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Source code segment from <code>refresh_potential</code> function of <code>mcf</code>.</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>An excerpt of modified <code>mcf</code> benchmark</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>New hardware tables</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>The relative performance of data-triggered threads for the CMP and SMT configurations, relative to our baseline. (HM means harmonic mean, and AM means arithmetic mean)</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>The execution time breakdown for our benchmarks for the CMP configuration.</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>The relative performance with CMP processor and single core configuration.</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>The performance with thread spawn latencies of 10 vs 500 cycles.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>The overhead of tracking changes to memory content and managing the internal data structures in Software DTT, with the support thread execution disabled.</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>The relative performance of the software DTTs implementation with both the main thread and the support thread functions running in a single thread.</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>The relative performance of the base software DTTs implementation with the support threads running on separate cores.</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>The relative performance of the software DTTs implementation with the fast thread spawning mechanism (DTT + fast spawn).</td>
<td>56</td>
</tr>
</tbody>
</table>
Figure 5.5. The relative execution time of DTT + fast spawn on the Nehalem machine. ......................................................... 57

Figure 5.6. The relative performance of the software DTTs implementation with fast thread spawning and thresholding mechanism (DTT + fast spawn + threshold), for SPEC. ........................................ 59

Figure 5.7. 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). ................................. 61

Figure 5.8. 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. ................. 64

Figure 6.1. An example of idempotent code ............................. 71

Figure 6.2. An example inter-procedural control flow graph .......... 74

Figure 6.3. The speedup of CDTT by using different silent store cutoffs running on the multi-core runtime system ................................. 80

Figure 6.4. The speedup of profile-assisted CDTT with different silent store cutoffs running on the runtime system with thresholding ........ 82

Figure 6.5. The speedup of CDTT on single-threaded and multi-core runtime system .......................................................... 85

Figure 6.6. The speedup of CDTT without profile (CDTT), profile-assisted CDTT with 20% silent store cutoff (profile-assisted CDTT) and programmers’ modification (hand-coded DTT) running on multi-core runtime system with thresholding .............................. 85

Figure 6.7. The runtime overhead of the runtime system that CDTT uses .... 89
ACKNOWLEDGEMENTS

I would like to acknowledge Professor Dean Tullsen for his support as the chair of my committee. Through my entire graduate school experience, including many papers and long nights, his guidance has proved to be invaluable.

I would also like to acknowledge Professor Steven Swanson for his kindly support during the worst of my PhD study and his key insights about this project. Without his support, this research may not be continued or published. His support helped me in an immeasurable way.

Thanks also to Professor Chung-Kuan Cheng, Professor Sorin Lerner, Professor Bill Lin for their valuable comments and feedback as my committee members.

I thank my collaborators and labmates in Dean Tullsen’s group and the Non-Volatile Systems Laboratory. Thanks for solving lots of technical questions and giving me insights in our research area.

I also want to thank my friends in UCSD, especially all my roommates and those who had dinner with me every Sunday evening. Without you, I cannot be mentally healthy all the time.

Finally, I want to thank my grandmother, to whom I owe everything in my life, for burying me into books since before I went to school, for engaging me into the beauty of computer science, for encouraging me pursue my dreams. I am sorry that I could not finish this sooner for you.

Chapter 2, Chapter 3, and Chapter 4 contain material from “Data-Triggered Threads: Eliminating Redundant Computation” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 17th International Symposium on High Performance Computer Architecture (HPCA 2011). The dissertation author was the first investigator and author of this paper. This material is copyright ©2011 by the Institute of Electrical and Electronics Engineers (IEEE).
Chapter 5, contains material from “Software data-Triggered threads” by Hung-Wei Tseng and Dean M. Tullsen, which appears in ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012). The dissertation author was the first investigator and author of this paper. This material is copyright ©2012 by the Association for Computing Machinery (ACM).

Chapter 6, contains material from “CDTT: Compiler-generated data-triggered threads” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 20th International Symposium on High Performance Computer Architecture (HPCA 2014). The dissertation author was the first investigator and author of this paper. This material is copyright ©2014 by the Institute of Electrical and Electronics Engineers (IEEE).
VITA

2003 Bachelor of Computer Science, National Taiwan University
2005 Master of Computer Science, National Taiwan University
2007–2014 Research Assistant, University of California, San Diego
2011 Candidate of Philosophy, University of California, San Diego
2014 Doctor of Philosophy, University of California, San Diego

PUBLICATIONS


This thesis introduces the data-triggered threads (DTT) programming and execution model. Unlike threads in conventional parallel programming models, the DTT model initiates threads on changes to memory locations. This enables increased parallelism and the elimination of redundant, unnecessary computation.

This thesis shows that 78% of all loads fetch redundant data, leading to a high incidence of redundant computation. By expressing computation through the DTT model, that computation is executed once when the data changes, and is skipped whenever the data does not change. The set of C SPEC benchmarks show performance speedup of up
to 5.9X, and averaging 46% with architectural support.

To improve the generality of the DTT model, this thesis also demonstrates a software-only runtime system that allows DTT programs running on top of existing machines. With mechanisms to minimize the multithreading overhead and dynamically turning on/off the DTT model, 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. We also show that the DTT model can work in conjunction with traditional parallelism using the software-only framework. The DTT model provides up to 64X speedup over parallel applications exploiting traditional parallelism.

This thesis also discusses CDTT, a compiler framework that takes C/C++ code and automatically generates a binary that applies the DTT model to eliminate dynamically redundant code without programmer intervention. With the help of idempotence analysis and inter-procedural name dependence analysis, CDTT identifies potential code regions and composes support thread functions that execute as soon as live-in data changes. CDTT can also use profile data to target the elimination of redundant computation. The compiled binary running on top of a software runtime system can achieve nearly the same level of performance as careful hand-coded modifications in most benchmarks. CDTT improves the performance of serial C SPEC benchmarks by as much as 57% (average 11%) on a Nehalem processor.
Chapter 1

Introduction

The ubiquity of computing resources lead us to an era of data explosion. As of 2012, we created an average of 2.5 exabytes of new data every day through numerous computing devices, social networking applications, online services, scientific computing, and online business transactions [1]. IDC projects that the trend of daily data generation will exponentially increase by 50X from now to the end of this decade [23]. Building computer systems and applications that match up the performance demand of transforming this rapidly growing application data becomes increasingly important. However, the Von Neumann model [54] that drives most computers today can limit the parallelism and create uncessary computation when processing application data [6, 7].

The Von Neumann architecture stores instructions and data in the memory units and uses processing units to execute instructions. The processing unit in a Von Neumann machine contains instruction registers and a program counter. At each step of executing a program, the processing unit fetches and executes the instruction where the program counter is pointing. The program counter either advances to the next instruction, or a new program counter is calculated (e.g. as the result of a branch instruction) after an instruction finishes. This is also true for parallel architectures based on the Von Neumann model. Typically, the computer starts parallel computation (i.e., threads) when the program counter reaches a fork or maybe a pthread_create call. Even for helper
```c
int S[N];
int D[N];

int update() {
    int i;
    .
    for(i=0;i<N;i++) {
        S[i] = getInput();
    }
    .
    for(i=0;i<N;i++) {
        D[i] = compute(&S[i]);
    }
    .
    return 0;
}
```

**Figure 1.1.** The concept of the DTT model

thread architectures [15, 16, 56, 55] or thread level speculation [36, 46, 48], it still requires the program counter to reach a trigger instruction or a system call before a new thread is spawned or activated.

Exploiting parallelism in the program counter based execution model of the von Neumann architecture is challenging because it requires the programmer to explicitly partition the computation running on top of different hardware contexts. In many cases, the algorithms exhibit limited potential for parallelism in the conventional model [28]. In addition, this traditional architecture is unaware of the working data. Therefore, the computer can potentially generate a significant amount of redundant computation that repeats the same operations. We find that in the C SPEC benchmarks, 78% of loads are redundant (meaning the same load fetches the same value as the last time it went to the same address). The computation which operates on those values is often also redundant.

This thesis proposes data-triggered threads (DTT), a programming and execution model, to address the deficiencies of the Von Neumann model. In contrast to the Von Neumann model, the DTT model initiates computation when the program changes...
particular memory contents. This has two primary advantages. First, computation depending on the changing data can execute immediately regardless of the position of the program counter, often exposing parallelism earlier and providing new opportunities for parallelism. Second, untouched or unchanged data do not generate unnecessary computation. The latter effect, eliminating unnecessary, redundant computation, is shown to be particularly powerful.

Figure 1.1 illustrates the concept of the DTT model. The C code in Figure 1.1 contains two for-loops. The first for-loop updates each element of array $S$. The second for-loop updates array $D$ by executing the `compute` function, which calculate the result using only the value from an array element, on array $S$. In this example, the computer completely recalculates every element of array $D$, even though $S$ may have only changed slightly, or even not changed at all.

In the DTT model, you can specify that $D$, or specific elements of $D$, are recalculated as soon as an element of $S$ changed. For example, if the statement $S[i] = \text{getInput}();$ changes the value of $S[0]$, the DTT model can immediately initiate a thread to recompute the value of $D[0]$ in parallel. In this way, the DTT model exploits parallelism at the earliest possible point in the program – when the data is generated or changed.

On the other hand, if the value of $S[2]$ remains the same after the statement $S[i] = \text{getInput}();$ is executed, recalculating $D[2]$ is unnecessary since the result will not change. In this case, the DTT model can avoid the redundant computation by triggering no computation in updating $D[2]$.

Because of the design of the DTT model, the computer can support this model with only a relatively small amount of architectural changes comparing with existing works sharing similar goals. For example, because the DTT model does not speculatively spawn threads, the DTT model does not require additional hardware or software
data structures to keep track of different versions of data or squash threads like speculative multithreading [36, 46, 48, 8] or value prediction [34]. Because the DTT model directly compare the sameness of data, the DTT model does not employs large hardware tables that are required by dynamic instruction reuse [47], block reuse [25], and silent stores [33] in book-keeping input-output pairs.

In addition, we show that supporting the DTT model using a software-only approach is feasible. The software-only runtime system makes the DTT model available on existing machines and improves the generality of the DTT model. Without hardware support, the software-only approach results in some runtime overhead. However, we will show that in many cases, the overhead is negligible comparing with the huge benefits that DTT model brings. Because the DTT model exploits redundancy, the DTT model not only skip the redundant execution, but also often miss the software overheads of tracking the data values and managing threads. As a result, the software solution can in many cases be competitive with the hardware version. The software-only DTT framework achieves an average of 15% performance improvement. We also demonstrates that the DTT model is complementary with traditional parallelism and can speed up an application by 64X.

The initial proposals of the DTT model rely on programmers to identify the potential of exploiting parallelism and redundant computation. However, automatically applying this model to existing programs is also possible. With the help of idempotence analysis and inter-procedural name dependence analysis, the compiler can identify potential code regions and generate binaries compatible with systems supporting the DTT model. The experimental result not only shows the performance matches programmer’s modifications, but also demonstrates that the algorithm for identifying code regions for the DTT model serves as a good static predictor for redundant code. Without any programmer’s intervention, CDTT compiler can still improve performance by 10% on the
software runtime system. The algorithm of selecting code regions to apply the DTT model also serves as an accurate static predictor of dynamically redundant code.

This thesis describes and evaluates all of the above aspects of the DTT model. We organize this thesis in the following way.

Chapter 2 demonstrates that nearly 80% of load instructions and more than half of all computation in a set of SPEC2000 applications could be redundant.

Chapter 3 presents the DTT model. We first show how the DTT model can execute a program, trigger parallel computation, and avoid redundant computation. We also introduce the programming paradigm of the DTT model – we proposed an extension to the C/C++ programming languages.

Chapter 4 examines the performance benefit of the DTT model using architectural support. This chapter shows that supporting the DTT model requires a small set of changes in the instruction set architecture and the microarchitecture. With minor programmers’ changes to SPEC2000 benchmarks using the DTT model, these changes can achieve up to 5.89X speedup and an average of 1.46X speedup.

Chapter 5 presents a software-only runtime system that supports the DTT model without hardware changes. This chapter also identifies potential performance bottlenecks in supporting the DTT model in software and presents optimizations minimizing multithreading overhead and automatically adjusting the usage of the DTT model.

Chapter 6 presents CDTT, a compiler that can automatically generate data-triggered threads from legacy code. The CDTT compiler further releases the burden of the programmer when applying the DTT model to existing applications. CDTT automatically identifies the code regions that can apply the DTT model and produces binaries running on top of systems supporting the DTT model.

Chapter 7 places the DTT project in context by discussing the related work and Chapter 8 concludes this thesis.
Chapter 2

Redundant computation

In a simplified model of execution in the Von Neumann architecture, we can consider a program as many strings of computation that begin with one or more loads of data – and are followed by computation on that data using registers, and completing with one or more stores to memory. If the data loaded remains the same since the previous invocation of this code, it is likely the computation produces the same results, and the program stores the same values to the same locations in memory.

Consider the for-loop in Figure 2.1 that we excerpt from the gcc compiler. If $j = 1$ and the content in `bb_live_regs[1]` remains the same from the last time we executed this for-loop, the statement `live = bb_live_regs[j];` will load the same data from `bb_live_regs[1]`. As a result, the statement `old_live_regs[j] = live;`

```c
for (j = 0; j < regset_size; j++)
{
    REGSET_ELT_TYPE live = bb_live_regs[j];
    old_live_regs[j] = live;
    if (live)
    {
        ...
        ...
    }
}
```

**Figure 2.1.** A code example of redundant computation from gcc
will produce the same result to old_live_regs[1]. If the bb_live_regs array only changes slightly every time we execute the for-loop, most of the load and store instances in this for-loop will reproduce the same result as the last invocation of the loop.

Lepak and Lipasti [32] demonstrated that 20-68% of all stores are silent stores. They define a silent store as a store instruction writing the same value already present at the memory address. For example, an instance of old_live_regs[j] = live; is a silent store if the value in oldLive_regs[j] is equal to live. By identifying silent stores and squashing silent stores in the memory queue, the processor can reduce the store traffic to the memory hierarchy and improve performance by 10% [33]. In this thesis, we are more interested in the other end, redundant loads, because we want to skip the entire string of computation leading to the silent stores, not just the store. Therefore, we also study the incidence of redundant loads.

We define a redundant load as a load instruction where the last time this load loaded this address, it fetched the same value. If the content in bb_live_regs[1] remains the same from the last time this for-loop completed, the statement live = bb_live_regs[j]; will incur a redundant load instance when j = 1. This redundant load also results in a silent store in the later code if old_live_regs[j] is not modified elsewhere.

This definition of the redundant load is different from that studied or exploited by existing works including value prediction [34, 35, 12]. Value prediction only considers a load instruction as redundant if the results of two consecutive instances of the load instruction are the same. In the code example of Figure 2.1, they can only cover the cases where bb_live_regs[j] and bb_live_regs[j-1] are the same, but may not include the redundant load that we define in this thesis. In addition, a redundant load in the prior definition does not necessarily lead to a silent store. If old_live_regs[j-1] is different from old_live_regs[j], the store operation cannot be skipped even though
Figure 2.2. The redundant load and silent store instructions in SPEC2000 C benchmarks

the loads from bb_live_regs return the same value. Therefore, these works can only
speculatively pre-execute some instructions, but not skip the computation string.

Using the above definitions of redundant loads and silent stores, we modified the
SMTSIM [52], an execution-driven, cycle-accurate simulator, to capture the percentage
of redundant loads and silent stores in the SPEC2000 C benchmarks. We compiled these
benchmark using the C compiler on an Alpha machine with -O2 optimization. Figure 2.2
shows the experimental result. We found that 79% of all loads are redundant, ranging
from 63% to 87%. Nearly all executed instructions depend (directly or indirectly) on at
least one load and thus inherit much of that redundancy. As a result, these instructions
perform the same computation as the last time them interact with the same input data.
As a result, over 50% of all stores are then silent.

For the code example in Figure 2.1 that contains the second most frequent stores
in the gcc program, 70% of the loads in the loop are redundant, and these loads result
in 70% silent stores. Even for ammp that has only 11% silent stores, we can still find a
computation string that incurs about 99% redundant loads and silent stores in the most
time-consuming function.
Acknowledgements

This chapter contains materials from “Data-Triggered Threads: Eliminating Redundant Computation” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 17th International Symposium on High Performance Computer Architecture (HPCA 2011). The dissertation author was the first investigator and author of this paper. This material is copyright ©2011 by the Institute of Electrical and Electronics Engineers (IEEE).
Chapter 3

The data-triggered threads model

Data-triggered threads 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. Chapter 2 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 brings in redundant computation that depends on that load, typically leading to a store that finally writes a value into memory that has not changed. With the DTT model, the programmer can express the redundant computation in a support thread, which is only triggered when the data has actually been changed.

This chapter will describe how the DTT model executes a program and discuss how a programmer can write a program using the DTT model.
Unlike traditional execution models, which generate parallelism based on control flow, the DTT model initiates a thread when the program changes particular memory contents. Figure 3.1 shows the basic operation of the data-triggered threads execution model. The original application contains code sections A, B, and C. The computation in code section B depends on data generated by code section A. If the store instruction in code section A generates a value different from what is currently stored in memory, the system will spawn a support thread (B’) using a free hardware context. The spawned thread performs the computation or an incremental version of section B in non-speculative fashion.

After the support thread (B’) completes execution and the main thread reaches the original location of section B, the processor will skip the execution of section B and jump to section C. This is possible because the computation of section B was done in B’ with exactly the same input data.

If the store instruction does not modify the value stored in memory or the program does not execute the store instruction, the computation and memory access operations of section B are redundant. In this case, the DTT model will not spawn a thread.
but skip the instructions in section B to eliminate the redundant computation. Because the DTT model skips the execution of code section B if the result of a prior support thread computation is still valid, the DTT model calls the code section B the “skippable region”.

Although in the common case the code in B will either be pre-executed or unnecessary, we leave the code in place (delineated by pragmas). This is done for two reasons. (1) The support thread may have failed to spawn, and (2) the support thread may have encountered an unexpected code path which caused it to squash itself. Either of those cases will result in the original B code executing in place.

3.2 The data-triggered threads programming model

To describe a program that can initiate computation using the data-triggered threads model, we propose a set of extensions of the C/C++ programming languages. These language extensions allow the programmer to exploit the DTT model by declaring data triggers, associating each data trigger with a support thread function, and defining the skippable region.

3.2.1 Specifying data triggers

Data-triggered threads are triggered by modifications to data. Therefore, the triggers themselves do not appear in the code section of the program, but rather in the data declarations. In the basic DTT programming model, the programmer can declare a trigger in two places, in a structure declaration or a variable declaration.

For each data structure field or variable that may spawn a thread, the programmer can attach the pragma #trigger to the definition of the data structure or variable. The format of the pragma is:

#trigger support_thread_function_name()
node_t *net; #trigger refresh_potential_DTT();

(a) uses a trigger attached to a specific variable

typedef struct node {
    ...
    struct node *pred; #trigger refresh_potential_DTT()
    struct arc *basic_arc; #trigger refresh_potential_DTT()
    cost_t potential; #trigger refresh_potential_DTT()
    long orientation; #trigger update_checksum_DTT()
    ...
} node_t;

(b) uses a trigger attached to a particular field of a defined data structure

Figure 3.2. Two possible ways to specify triggers.

In a variable declaration, we are specifying that the thread be initiated any time that variable is changed (Figure 3.2(a)). In this way, we can attach a support thread function to specific variables of a type. If the programmer declares an array as a data trigger, the DTT model will trigger the support thread function when any element in the array changes.

On the other hand, when we declare a trigger in a structure declaration, we can attach the support thread function to a particular element of the structure (Figure 3.2(b)). This enables us, for example, to spawn a thread when we touch the potential field of a node_t structure, but not when we touch another field unrelated to this calculation.

If the programmer declares a variable or a data structure field as a data trigger, the compiler will replace all store instructions that may touch the memory content of the data trigger with a special instruction or a software function, described in the following chapters, to detect the change of the memory content. The special instruction or function can initiate computation if it detects changes in the memory content.
We would like to avoid a flurry of unnecessary threads when data structures that may otherwise be very stable are initialized. Therefore, we also allow the programmer to use the `#notrigger` pragma to identify a region of code for which we do not want to spawn a thread, even if it modifies one of our data triggers.

Because we depend on the compiler (rather than hardware that watches for an address) to identify trigger locations, this impacts what accesses we can track. If we pass a pointer to a trigger variable to a function, for example, then modify the variable through that pointer, a thread will not be generated. However, if we specify the trigger in a structure definition, it will cause the thread to be spawned in this case, as long as the type of the pointer is declared correctly in the callee function.

### 3.2.2 Composing support thread functions

The support thread function describes the computation to perform after the memory content of a data trigger changes. The support thread function is written as a function in a conventional language and accepts the triggering address as an input. When executing the support thread function, the spawned thread occupies the same address space as the main thread, and can read and write anything in memory. The support thread function also has its own stack so that it can maintain its own local variables and even make function calls.

Clearly, we do not want the data-triggered threads to create unwanted data races when executing the support thread function in parallel. As with any parallel code, it is up to the programmer to determine that those stores do not create data races. Because the timing of the writes is always constrained to be between the triggering point and the main thread join point (skippable region), it is often easy to verify the absence of data races.

There are some restrictions when composing support thread functions. First, the
support thread function should be *idempotent* [19]. A code segment is idempotent if the code does not overwrite any of its inputs. As a result, this code segment always generates the same result given the same inputs. This also implies that the result is not affected by how many times the code is executed – as long as the input values remain the same. In the DTT model, threads are started and potentially aborted asynchronously, and may be executed multiple times before the result is used. Therefore, a thread that (for example) accumulates state each time it executes will not be 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.

Sometimes, the programmer realizes that the result of a triggered thread cannot be reused when the thread takes a particular path – for example, if an unlikely condition is met, causing the code to potentially access some data that may still be changing (e.g., a trigger had not been applied to that data for some reason) or may create a race condition. In this case, the programmer can use the cancellation feature to invalidate the status table entry and stop the current thread by adding the pragma

```c
#pragma cancel
```

in the code segment. This guarantees that the code in the corresponding skippable region will be executed again by the main thread before the result is accessed.

### 3.2.3 Defining skippable regions

In our current instantiation of the data-triggered threads programming model, (often identical) code appears in two places: in the definition of the support thread function, but also in the main thread. If you are modifying existing code (as in all the applications considered in this paper), think of the latter as the place where the original code region was before modification. This second copy of the code serves several purposes. First, it serves as the join point of the main thread and the support thread, in the case
where the support thread successfully pre-executed. Second, it serves as the backup in case the support thread did not spawn or did not complete successfully. In those cases, the main thread can just execute that code in place. Otherwise, the code is skipped, and the return value (if any) is copied into a register.

To define the boundaries of these skippable regions in the main thread, the programmer needs to add pragmas into the program source code. The pragma

```
#block block_name
```

defines the beginning of the skippable region, and the pragma

```
#end_block
```

the end of the skippable region. In each support thread function, the programmer uses the pragma

```
#DTT block_name
```

to specify the corresponding skippable region that can be skipped after all spawned support thread functions complete.

In the DTT model, we allow more than one support thread functions to replace the computation of a single code block. Consider this example: the potential field and an orientation field could each trigger separate thread functions, but each would specify the same block_name in the pragma, allowing each to replace the piece of code that recomputes multiple elements of the tree network.

### 3.2.4 Code example

To provide an overview about how to apply the DTT model in a program, we use the most time-consuming function in mcf, the refresh_potential function as an example in this section. Figure 3.3 shows a while-loop in the refresh_potential function. Our experiment in Chapter 2 shows that more than 99% of the computation in this function is redundant.
while( node ) {
    if( node->orientation == UP ) {
        node->potential = node->basic_arc->cost +
        node->pred->potential;
    } /* == DOWN */
    else {
        node->potential = node->pred->potential -
        node->basic_arc->cost;
        checksum++;
    }
    tmp = node;
    node = node->child;
}

Figure 3.3. Source code segment from refresh_promise function of mcf

The while-loop in the refresh_promise function updates the potential of all nodes in a network. For each node, the value of the potential field only depends on the potential field of the linked node pred and the cost field of the edge basic_arc. In other words, the value of potential changes only when the program changes the value of potential or cost of its pred and basic_arc nodes, or when the node has different pred and basic_arc nodes.

Because the interactions are complex, explicitly tracking changes and their implications would be difficult. However, these particular fields (including the links) of this structure change slowly, so the code constantly recalculates the same potential values; that is, the loads are redundant and the computation and stores are unnecessary. In this implementation, using traditional programming features, the amount of computation is constant, regardless of how much or how little the data in a network changes.

To generate the same result as the original code but also avoid redundant computation, the programmer can use the DTT model to rewrite the code as in Figure 3.4. We use the second mechanism (Figure 3.2(a)) for declaring data triggers. The second mechanism is more efficient in this code because only modifications to the potential and the cost fields and pointers to linked nodes (e.g., basic_arc and pred) can change the value of potential. Figure 3.4(a) shows our data trigger declarations. Whenever one of these fields changes, the DTT model can initiate the execution of the
typedef struct node {
    struct node *pred;  //trigger refresh_potential_DTT()
    struct arc *basic_arc;  //trigger refresh_potential_DTT()
    cost_t potential;  //trigger refresh_potential_DTT()
    long orientation;  //trigger update_checksum_DTT()
} node_t;

(a) data trigger declaration

DTT refreshPotential

void refresh_potential_DTT( node_t *root ) {
    node_t *node, *tmp;
    tmp = node = root->child;
    while( node != root ) {
        while( node ) {
            if( node->orientation == UP )
                node->potential = node->basic_arc->cost +
                node->pred->potential;
            else /* == DOWN */
                node->potential = node->pred->potential -
                node->basic_arc->cost;
            tmp = node;
            node = node->child;
        }
        node = tmp;
        while( node->pred ) {
            if( node->orientation == UP )
                node->potential = node->basic_arc->cost +
                node->pred->potential;
            else /* == DOWN */
                node->potential = node->pred->potential -
                node->basic_arc->cost;
            tmp = node;
            node = node->child;
        }
    }
}

(b) the support thread function

long refresh_potential( network_t *net ) {
    node_t *stop = net->stop_nodes;
    node_t *node, *tmp;
    node_t *root = net->nodes;
    root->potential = (cost_t) -MAX_ART_COST;
    tmp = node = root->child;
    while( node != root ) {
        //block refreshPotential
        root->potential = (cost_t) -MAX_ART_COST;
        tmp = node = root->child;
        while( node != root ) {
            //end_block
            return checksum;
        }
    }

(c) modified refresh_potential function

Figure 3.4. An excerpt of modified mcf benchmark
refresh\_potential\_DTT function in a support thread. This is a particularly effective construct; in this case, the contents of nodes may change frequently, but the structure and the value of the potential or cost fields may not – this allows us to ignore all but the exact changes we care about.

Figure 3.4(b) presents the support thread function that executes when any of these fields in a node changes, the refresh\_potential\_DTT function. This support thread function updates the potential fields of every nodes in the subtree that is rooted at the changed node. This support thread function only updates the modified node and its succeeding nodes because the change of a node never affects the potential fields of its predecessors. This implementation of the support thread function allows us to avoid more redundant computation.

If the value of the potential field in any node is not going to change further and all the support thread functions triggered by modified nodes completed, the program does not need to compute the code in the while-loop again. Therefore, we defined the while-loop as the skippable region as in Figure 3.4(c). When the main thread reaches the original refresh\_potential function, the DTT model will skip the execution of that function, because all the values that need to change will have already been computed and written.

We could also try to apply a software technique like memoization [37, 14] to this code. Because it depends on a global linked list of unknown size, it would require virtually unlimited storage for old values, and the cost of checking the input structure for sameness and transferring all of the saved output values is nearly the same as the computation itself. With data-triggered threads, we store only a few bytes, independent of the size of the live-in data structures, and completely bypass the sameness check, memory value copy, and the computation.
Acknowledgements

This chapter contains material from “Data-Triggered Threads: Eliminating Redundant Computation” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 17th International Symposium on High Performance Computer Architecture (HPCA 2011). The dissertation author was the first investigator and author of this paper. This material is copyright ©2011 by the Institute of Electrical and Electronics Engineers (IEEE).
Chapter 4

Architectural support for the DTT model

The computer system can support the DTT model either with hardware and the instruction set architecture (ISA) or a software-only runtime system. This chapter describes the former option. Chapter 5 will describe the software-only option.

To support the DTT model, the computer must be capable of detecting changes in the memory, executing support thread functions in parallel, and skipping the unnecessary computation in the skippable region. In the hardware solution presented in this chapter, we leverage the existing multithreading processors to perform the support thread functions, modify the ISA to detect memory content change, and use hardware tables to guide the execution of skippable regions.

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.

On the ISA side, the DTT model proposes several new instructions: \texttt{tstore}, \texttt{tspawn}, \texttt{tcancel}, and \texttt{treturn}. The \texttt{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 \textit{tstore} instruction detects a change to memory contents, the following \textit{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 \textit{treturn} instruction that updates the TST, or a \textit{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 \textit{tstore} instruction also assumes changes to the load/store pipe and cache to determine whether the value has been changed in memory.

This chapter will describe this modest architectural support to enable the DTT model. This chapter will also show that existing, complex code can be transformed to exploit the DTT model with minor changes. This chapter will also present the performance gains from these transformations, which can be as high as 6X speedup.

4.1 Architectural support

The DTT execution model assumes processors capable of running multiple hardware contexts, such as a chip multiprocessor or simultaneous multithreading. However, it also works on a single-thread core with software threads. To support the DTT model, we propose some architectural changes to the baseline processor. These include the \textit{thread status table} and the \textit{thread queue}. In addition, a set of new instructions, \textit{tstore}, \textit{tspawn}, \textit{tcancel}, and \textit{treturn} are added to the existing instruction set architecture.

In this section, we will introduce these new architectural components and discuss how they enable the data-triggered threads model.
4.1.1 ISA support

For the hardware implementation of data-triggered threads, we propose four new types of instructions in the instruction set architecture. This implementation also requires compiler modifications to generate binaries that can utilize the new instructions.

The primary addition in the ISA is the \texttt{tstore} instruction. This type of instruction causes a thread to be generated if the store modifies memory. We also considered full hardware solutions for tracking changes to memory values (e.g., a table that watches memory addresses or regions); however, the ISA solution we use here has several key advantages. (1) It greatly simplifies triggering based on specific data fields. In the \texttt{refresh.potential} example, we can trigger on a change to the \texttt{pred} or \texttt{basic.arc} fields of a node, but ignore changes to other fields — we could not do this with hardware that tracked changes to a region of memory. (2) It makes it easier to ignore some accesses (such as initialization of the structure) by just not using the \texttt{tstore} instruction. (3) It allows us to track a larger set of addresses, not constrained by the size of some internal table.

Whenever the main thread executes and commits a \texttt{tstore} instruction, hardware detects whether the store is silent or not. A good description of the hardware to detect silent stores is in [33]. Note that we do not simulate the performance gains from short-circuiting silent stores except for \texttt{tstore} instructions, to better evaluate the impact of our proposal in isolation. If the store does modify memory, we cause a thread to be spawned by creating a support thread event in the \texttt{thread queue}.

To associate a \texttt{tstore} instruction with a support thread function, we must follow it with a \texttt{tspawn} instruction, which fills the two hardware tables, \texttt{thread queue} and \texttt{thread status table}, with the the PC of the support thread function, the \texttt{start PC} of the skippable code in the main thread, the \texttt{destination PC} which denotes the new PC after the region
The code executed in the support thread function has some implicit arguments (which become live-ins) and a return value. These live-ins include the global pointer, stack pointer, and the triggering address. At most, there will be one register live-out if the function returns a value, zero if not.

The hardware injects move instructions right after the \textit{tstore} triggers a thread to transfer the implicit live-ins (sp, gp) from registers to the Thread Queue to create a support thread event. We currently have two kinds of \textit{tstore} instructions: for the declaration in Figure 3.2(a), we pass the effective address to the TQ. For the structure-based declaration in Figure 3.2(b), we pass the base address of the structure to the TQ. In this case, we just need to ensure that the compiler constructs \textit{tstore} instructions carefully (and conventionally), with the base register containing the base address of the structure and the displacement field the offset – in that case the base address in the register, not the computed address, is inserted into the thread queue. If the \textit{tstore} does not alter memory (not actually known until the instruction commits), the \textit{tspawn} instruction is ignored and the transfers do not take place. Because our results in Section 4.3 indicate that performance is highly insensitive to thread spawn costs, we could transfer the implicit live-ins through memory via software and it would perform about the same.

When the support thread function encounters a \textit{tcancel} instruction, the thread will terminate its execution immediately. As discussed previously, this enables us to create a support thread function in a case where an infrequent live-in is still not calculated. In this case, we cancel the thread if we take a path that would read the unexpected value.

When the support thread function executes a \textit{treturn} instruction, the processor will finish execution of the current support thread. The current value of the thread live-out will be copied into the TST – this may involve remote communication, depending upon the location of the TST.
Figure 4.1. New hardware tables

In order to maximize the exposure of redundant execution, we could have a variant of the `tspawn` and `treturn` that are executed by the main thread when it executes its version of the data-triggered threads code. This would allow its live-out to be written into the TST to bypass future redundant computation.

4.1.2 New hardware structures

To efficiently support the DTT model, we add the thread queue (TQ) and the thread status table (TST) as shown in Figure 4.1.

When a thread executes a `tstore` instruction that modifies memory, we will create a new entry in the TQ, as described above, with data from the `tstore` and the thread registry. The TQ holds the start PC and arguments for any thread that has been requested but not yet completed. At the same time, we also allocate or modify an entry in the TST corresponding to this thread, filled with data from the TQ and `tspawn` instruction – in particular the start PC (the beginning of the skippable region) and destination PC (the first instruction following the skippable region).
Each TST entry also contains a location for (and register name of) the register live-out, if the thread has one. This value is written when a support thread function completes. In addition, each entry in the TST contains status bits to indicate if this entry is valid, invalid, spawning, or running. If a new event enters the TQ, the corresponding TST entry will change to spawning state.

When a hardware context is available and the TQ is not empty, a thread is spawned with the PC and arguments based on the values stored in the TQ. However, if there is a TST entry corresponding to the same code block when a thread is triggered, and its status is running, one of two things will happen. If the triggering address is the same, the running thread is aborted in favor of the new thread. If the triggering address is different, the new thread waits to spawn until after the first completes — this is a conservative approach and may not be necessary in all cases. If there is no available context when a thread is ready to spawn, the TST entry is simply marked invalid, ensuring that the computation will be performed by the main thread.

When the main thread’s PC reaches a value that matches a start PC entry that is valid, a register move instruction is injected into the pipeline to move the register live-out from the TST to the local register file. Then the PC is changed to the destination PC value. When code is highly redundant, we will do this latter operation much more often than we will spawn threads.

If the TST status bits specify invalid, we will just execute the code in place. If the status bits indicate that a support thread event is still spawning, we will remove the support thread event from the TQ and execute the code in place, and the support thread event will never execute. If the status is running, we will stall the main thread until the entry becomes either valid or invalid. We can either have one TST per core, or have it centralized. In the latter case, we can exploit redundancy between parallel threads more easily, but would need to cache at least part of the TST data in each fetch unit for
fast comparison with the current program counter each cycle. We assume one TST per core, with remote threads communicating register output values to the TST across the interconnect when the support thread returns.

Our TST currently allows one entry per code block (identified by the PC of the code in the main thread). This does not change the programming model, but may limit the performance. Consider the case of a routine that calculates the determinant of a matrix. If it is always called for the same slowly-changing matrix, it will detect the redundancy. If it is called for 5 different slowly-changing matrices, it will not identify the redundancy when it is called for a different matrix than the last call. This is because the arguments to the function will differ from the previous call. This is an implementation detail that can be changed in future implementations. It was not a major impediment to the current set of applications. The primary impact was that we sought out very coarse-grain threads that operate on entire data structures, rather than fine-grain threads that made local changes in reaction to writes to individual elements.

In summary, we add only a few small tables, accessed infrequently. The only frequent access is the comparison of the skippable region start address with the program counter, a comparison similar to, but less complex than, the BTB access. Thus, we add no significant complexity to the core. We do add some hardware to the ECC check circuit of the L1 data cache – it is the same hardware proposed for silent stores [33, 32] which incurs no extra delays.

4.2 Experimental methodology

4.2.1 Opportunities for data-triggered threads

Data-triggered threads can enhance parallelism by starting dependent computation as soon as the source data is changed, or to reduce unnecessary computation by
placing often-redundant code in a thread. For this initial investigation of these ideas, we focus on the latter, but do exploit the former when it presents itself in the neighborhood of the redundant computation we are studying. We profile SPEC C applications for procedures or code regions with high incidence of redundant loads. For each application, we then identified just one or a few regions to investigate, making relatively minor changes to the code.

However, not all code is a candidate for data-triggered threads. Code where the data is changed very close to the original code region will not create parallelism, but can still pay off if the redundancy is high. Conversely, code based on frequently changing data is only a good candidate if it can be triggered well ahead of time to exploit parallel execution. Data that often changes *multiple times* between invocations of the targeted code region will still work, but can create extra work and may not perform well.

One new constraint that C programmers may not be used to is that each support thread function must be idempotent, as described in Chapter 3. Because threads are started and potentially aborted asynchronously, and may in fact be executed multiple times before the result is used, a thread that (for example) accumulates state each time it executes will not be a good candidate. If the `refresh_potential` function, from Figure 3.3, accumulated the total potential and added it to a running sum, it would not work without some restructuring. But because it only writes local variables that are initialized in the routine or global variables that will be rewritten on a restart, it is an excellent candidate. We expect the compiler to help to identify and flag support thread functions that are not idempotent.

For this set of experiments, we have modified all of the C SPEC2000 programs – our current framework only works with C code. In each case, we identify no more than three routines that appeared to be good candidates for a data-triggered thread, and modify the code accordingly. The changes to the source code were extremely minor in
all cases. For example, in mcf, we copied part of the subroutine refresh_potential (35 lines), added two lines of code to prevent redundant computation, and also added 7 pragmas. Table 4.1 lists the number of static instructions of our DTTs. The average length of our DTTs is 145 instructions.

We summarize our modifications to the C SPEC 2000 benchmarks in Table 4.1. We also provide more detail regarding our implementation in Chapter A. We were guided heavily by the profile data regarding functions and code regions with high incidence of redundant loads. We exploited opportunities to increase parallelism only when they presented themselves during that process — we did not profile for opportunities for parallelism.

### 4.2.2 Simulation

In this set of experiments, we evaluate the architectural support for the data-triggered threads model using a modified version of SMTSIM[52]. SMTSIM is an execution-driven, cycle-accurate simulator which models a multicore, multithreaded processor executing the Alpha ISA.

We assume the baseline processor core is a 4-issue out-of-order superscalar processor with a 2-way 64KB L1 instruction cache and 2-way 64KB L1 data cache. The processor also has a 2-way 512KB L2 cache and a 2-way 4MB shared L3 cache. The global hit times of L1, L2, and L3 caches are 1 cycle, 12 cycles, and 36 cycles, and it takes 456 cycles to access main memory. The branch predictor used for simulation is a gshare predictor with 2K entries.

Since the DTT model will work with any processor capable of running multiple contexts concurrently, we tested our scheme on both chip multiprocessor (CMP) and simultaneous multithreading processors (SMT) [53]. We assume the CMP platform is a dual-core processor in which each core has a private instruction cache and data cache,
Table 4.1. Modifications to benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data triggers</th>
<th>Avg. static DTT inst.</th>
<th>Computation performed by data-triggered threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>last, naybor</td>
<td>193</td>
<td>Some code from a_number function to recompute total number of nodes and code from mm_fv_update_nonbon function to refresh atomall</td>
</tr>
<tr>
<td>art</td>
<td>f1_layer[].P</td>
<td>313</td>
<td>Some code from train_match function to refresh f1_layer[].Y</td>
</tr>
<tr>
<td>bzip2</td>
<td>ss</td>
<td>39</td>
<td>Computes the value of bbStart, bbSize and shifts</td>
</tr>
<tr>
<td>crafty</td>
<td>search</td>
<td>101</td>
<td>Some code from Evaluate function to precompute the new score</td>
</tr>
<tr>
<td>eon</td>
<td>a_MR and a_VHR in mrSurfaceList::viewingHit method</td>
<td>67</td>
<td>The constructor of the ggMaterialRecord or mrViewingHitRecord class</td>
</tr>
<tr>
<td>equake</td>
<td>time, disp[]</td>
<td>57</td>
<td>We trigger the computation of phi0, phi1, and phi2 functions once time changes. We also trigger a thread to perform the time integration computation when the smvp function generates a new value for a disp array element.</td>
</tr>
<tr>
<td>gcc</td>
<td>reg_rtx_no</td>
<td>4</td>
<td>The computation of max_reg_num function</td>
</tr>
<tr>
<td>gzip</td>
<td>strstrart, hash_head</td>
<td>30</td>
<td>The computation of longest_match function</td>
</tr>
<tr>
<td>mcf</td>
<td>node_t</td>
<td>35</td>
<td>Some code from refresh_potential function to update the subtree leading by the touched node</td>
</tr>
<tr>
<td>mesa</td>
<td>i0, wo, and w1</td>
<td>203</td>
<td>Generating new R, G, B, and alpha values</td>
</tr>
<tr>
<td>parser</td>
<td>randtable and inputs of count function</td>
<td>55</td>
<td>The computation of hash function</td>
</tr>
<tr>
<td>perlbmk</td>
<td>PL_op</td>
<td>46</td>
<td>Pre-executing the function specified by PL_op</td>
</tr>
<tr>
<td>twolf</td>
<td>new_total of dimptr</td>
<td>159</td>
<td>The computation of new_dbox_a function</td>
</tr>
<tr>
<td>vortex</td>
<td>EmpTkn010 and PersonTkn</td>
<td>168</td>
<td>The computation of PersonObjs_FindIn function</td>
</tr>
<tr>
<td>vpr</td>
<td>heap</td>
<td>30</td>
<td>The computation of my_allocate function</td>
</tr>
</tbody>
</table>
but shared L2 and L3 caches. The SMT processor can run at most two hardware threads – it is a single core with the same size caches as a single core on the CMP.

We also assume that there is an additional 10-cycle delay (unless specified otherwise) before spawning threads due to the transfer of register values. The TST contains 4 entries and the TQ contains 16 entries.

Because the data-triggered threads change the total number of dynamic instructions, we cannot use IPC as the performance metric. Instead, we set a check point within each benchmark (based on the Simpoint [45] and the desired simulation length) to compare the cycles each different configuration takes for the main thread to reach the checkpoint.

We use all 15 benchmarks written in C from both the SPEC CPU 2000 integer and floating point suites as our target benchmark suite, regardless of whether our profiling determined they were good candidates for data-triggered threads. This set of programs exhibit a wide range of data access behaviors including pointer deferencing and control flow behaviors. We simulated each benchmark for a total of 500 million instructions (based on the original code’s execution stream) starting at a point indicated by Simpoint [45]. All simulation results use the reference input sets. For each benchmark, we rewrote the functions containing the most redundant load instructions using the proposed DTT model as described in Section 3.2.2.

4.3 Results

Figure 4.2 shows the experimental results of our modified codes running on the DTT architecture. These applications achieve an average of 45.6% performance improvement over the baseline processor in the CMP platform. In the best case, we see a gain near 6X. Even the harmonic mean, which heavily discounts the positive outliers, shows an average gain of 17.8%.
Figure 4.2. The relative performance of data-triggered threads for the CMP and SMT configurations, relative to our baseline. (HM means harmonic mean, and AM means arithmetic mean)

The DTT model running on this architecture achieves a speedup of 5.89 on mcf. As discussed in Figure 3.4, we optimize the refresh_potential function, which traverses a large pointer-based data structure and incurs many cache misses. This is the most time consuming function within mcf, yet most of its computation is redundant.

For the SMT results, our architecture spawns support threads on another hardware context on the same core, possibly competing more heavily for execution resources. Even still, our DTT architecture achieves a 40% performance improvement on the SMT platform.

We further break down the execution statistics in Figure 4.3. This graph shows the percentage of cycles when only the main thread is running, when only the support thread is running, or both. We see that our largest gain, on mcf, comes completely from reduced execution. In the cases where we get no gains, the support threads just do not occur frequently enough (neither executed nor skipped). In a couple cases, we actually increases the total executed instructions. Due to parallelism effects, it is perfectly plausible that we could still get speedup when that happens – but in these cases we don’t.

Both the CMP and SMT implementations clearly benefit from the elimination
Figure 4.3. The execution time breakdown for our benchmarks for the CMP configuration.

Figure 4.4. The relative performance with CMP processor and single core configuration of redundant computation. The experimental results of the two architectures are nearly identical for benchmarks like mcf, mesa, vortex, and vpr. Table 4.2 shows that DTT reduces dynamic instruction counts by more than 10% for these benchmarks. However, the CMP has an advantage in exploiting parallelism, not having to compete for pipeline resources. For ammp, art, crafty, equake, and twolf, our DTT architecture helps to exploit parallelism between the main thread and support threads. But these benchmarks suffer from resource competition on SMT.

To further examine this point, we ran a non-multithreaded, single-core version of
Table 4.2. Relative number of dynamic instructions and cache miss rates for DTT configurations

<table>
<thead>
<tr>
<th>Name</th>
<th>Relative Num. of Dyn. Insts.</th>
<th>Cache Miss Rates</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DTT/CMP</td>
<td>DTT/SMT</td>
<td>Main Thread</td>
<td>DTT</td>
<td>baseline</td>
<td>DTT/CMP</td>
<td>DTT/SMT</td>
<td>baseline</td>
</tr>
<tr>
<td>ammp</td>
<td>0.65</td>
<td>0.51</td>
<td>0.65</td>
<td>0.32</td>
<td>3.26%</td>
<td>3.02%</td>
<td>3.84%</td>
<td>32.31%</td>
</tr>
<tr>
<td>art</td>
<td>0.68</td>
<td>0.34</td>
<td>0.68</td>
<td>0.34</td>
<td>31.09%</td>
<td>20.42%</td>
<td>31.56%</td>
<td>92.01%</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.01</td>
<td>0.00</td>
<td>1.01</td>
<td>0.00</td>
<td>1.23%</td>
<td>1.14%</td>
<td>1.15%</td>
<td>48.21%</td>
</tr>
<tr>
<td>crafty</td>
<td>0.96</td>
<td>0.09</td>
<td>0.96</td>
<td>0.07</td>
<td>1.07%</td>
<td>1.09%</td>
<td>1.24%</td>
<td>3.79%</td>
</tr>
<tr>
<td>eon</td>
<td>0.89</td>
<td>0.05</td>
<td>0.89</td>
<td>0.05</td>
<td>0.14%</td>
<td>0.14%</td>
<td>0.14%</td>
<td>1.84%</td>
</tr>
<tr>
<td>quake</td>
<td>0.60</td>
<td>0.30</td>
<td>0.60</td>
<td>0.30</td>
<td>9.32%</td>
<td>9.34%</td>
<td>11.88%</td>
<td>85.51%</td>
</tr>
<tr>
<td>gcc</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.78%</td>
<td>0.78%</td>
<td>0.78%</td>
<td>25.29%</td>
</tr>
<tr>
<td>gzip</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>4.61%</td>
<td>1.80%</td>
<td>1.80%</td>
<td>0.52%</td>
</tr>
<tr>
<td>mcf</td>
<td>0.56</td>
<td>0.00</td>
<td>0.56</td>
<td>0.00</td>
<td>36.92%</td>
<td>31.40%</td>
<td>31.32%</td>
<td>87.74%</td>
</tr>
<tr>
<td>mesa</td>
<td>0.87</td>
<td>0.01</td>
<td>0.87</td>
<td>0.01</td>
<td>0.34%</td>
<td>0.35%</td>
<td>0.37%</td>
<td>50.34%</td>
</tr>
<tr>
<td>parser</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>1.90%</td>
<td>1.86%</td>
<td>1.86%</td>
<td>41.51%</td>
</tr>
<tr>
<td>perlbnk</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.30%</td>
<td>0.30%</td>
<td>0.30%</td>
<td>6.98%</td>
</tr>
<tr>
<td>twolf</td>
<td>0.89</td>
<td>0.12</td>
<td>0.91</td>
<td>0.12</td>
<td>3.49%</td>
<td>3.50%</td>
<td>3.40%</td>
<td>50.31%</td>
</tr>
<tr>
<td>vortex</td>
<td>0.88</td>
<td>0.00</td>
<td>0.88</td>
<td>0.00</td>
<td>1.20%</td>
<td>0.99%</td>
<td>0.99%</td>
<td>16.66%</td>
</tr>
<tr>
<td>vpr</td>
<td>0.86</td>
<td>0.02</td>
<td>0.86</td>
<td>0.02</td>
<td>3.30%</td>
<td>2.96%</td>
<td>2.99%</td>
<td>51.23%</td>
</tr>
<tr>
<td>average</td>
<td>0.86</td>
<td>0.10</td>
<td>0.86</td>
<td>0.08</td>
<td>6.60%</td>
<td>5.27%</td>
<td>6.24%</td>
<td>39.62%</td>
</tr>
</tbody>
</table>
the architecture. With this architecture, support threads pre-empt the main thread when they are ready to run. This architecture exploits no parallelism, but again benefits from reduced execution due to redundancy. Those results are in Figure 4.4. Even that architecture achieved a 1.33 speedup despite a few benchmarks showing large slowdowns. If we tuned off DTTs for this case (e.g., not using DTTs when they cause slowdowns) the overall speedup would be 1.38.

Even though we were not targeting parallelism in general, we find that one reason that we do not expose as much parallelism as we might hope is our success at eliminating redundant computation. When the support threads rarely execute (e.g., mcf, vortex, vpr), they have little opportunity to execute in parallel.

We do see extensive parallelism in ammp, art, and equake. It is unexpected that we do not see significant SMT speedup in any of these cases. In ammp and equake, the parallelism does help to significantly improve performance in CMP, but incurs serious resource contention with the main thread in SMT – a single thread of ammp or equake uses almost all of the execution bandwidth of a single core. In art, eon, mesa, and twolf, the gains are mitigated in both the CMP and SMT cases because the main thread must often wait for the DTT. For CMP this tradeoff is a slight win, for SMT it is a clear loss. In an energy-conservative architecture, we would likely not use this approach for this benchmark.

There is another source of performance gain, however, besides parallel execution and fewer executed instructions. Table 4.2 shows that DTT helps to improve cache performance at all levels of the cache hierarchy. With a CMP processor, DTT reduces the L1 D-cache miss rate from 6.60% to 5.27% and L2 cache miss rate from 39.60% to 33.69%, on average. Even with the SMT processor, in which two threads compete for the shared L1 cache, DTT still reduces the L1 D-cache miss rate from 6.60% to 6.24% and L2 cache miss rate from 39.60% to 34.61%. Because we tended to target code that
traversed large data structures (because that’s where much of the redundancy was), DTT successfully eliminated code that had poor cache behavior.

The latency for spawning new threads is an important variable and will depend on several factors, including communication latencies between cores, the number of live-ins, etc. In our initial results, we model a very fast spawn latency to get an understanding of what gains are possible. To get a better feel for how the spawn latency affects performance, Figure 4.5 compares the performance of data-triggered thread under spawn latencies of 10 and 500 cycles. The performance does not make a significant difference in most of the benchmarks. In crafty, eon, twolf, and vpr, the data structure is frequently modified and there is insufficient slack between the execution of support thread functions and the code segment using the result; therefore, the high spawn overhead results in going from no gain to actually losing performance. In general, though, we lose little performance overall even with a high spawn latency.

4.4 Discussion

With the data-triggered threads model, the programmer can specify threads to spawn and execute when the application touches and changes data (of a certain variable,
or of a certain type). This enables increased parallelism, but in this initial foray into this execution model, the focus is on the elimination of redundant computation. This chapter presented that by specifying computation in the DTT model, the computation is only performed when the data gets changed, eliminating redundant executions. We presented that the DTT model only requires minor architectural changes. By making small modifications to existing C programs, we achieve speedups as high as 5.89, and averaging 1.46 with the architectural support of the DTT model.

Acknowledgements

This chapter contains material from “Data-Triggered Threads: Eliminating Redundant Computation” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 17th International Symposium on High Performance Computer Architecture (HPCA 2011). The dissertation author was the first investigator and author of this paper. This material is copyright ©2011 by the Institute of Electrical and Electronics Engineers (IEEE).
Chapter 5

Software data-triggered threads

In the previous chapter, we demonstrated that the DTT model can achieve up to 6X speedup, but requires changes to the instruction set architecture and the addition of hardware tables to the microarchitecture, making it unavailable on existing processors.

This chapter introduces a software implementation that supports the DTT model, software data-triggered threads (software DTT). Software DTT enables 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 chapter presents techniques for mitigating the software overheads of DTT. The thresholding mechanism, for example, addresses 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 DTT can be used to either add a new type of parallelism to serial code, or to augment traditionally parallel code. This chapter demonstrates both. 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).

5.1 Design and implementation of software data-triggered threads

In this section, we demonstrate the design of software DTT. Software DTT is 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 without hardware, the software-only system replaces the functionalities of the two hardware tables in Section 4.1, Thread Status Table and Thread Queue, using software data structures. We will describe these software data structures in Section 5.1.1.
Where we needed ISA changes in Chapter 4, we instead replace each of those new instructions with a function call; for example, the tstore instruction is now replaced with a call to tstore(), which is a library function in our software runtime system. In the software-only runtime systems, these function calls will play important roles in (1) detecting if an operation changes memory contents, (2) scheduling the thread, and (3) skipping over the skippable region, if appropriate. We will also explain these function calls in Section 5.1.2.

5.1.1 Software data structures

The hardware framework to support the DTT model uses the Thread Status Table and Thread Queue (Section 4.1) 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 5.3.4. Even then, we use no more than 40 bytes of memory for each state variable. In this chapter, 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 chapter.

5.1.2 Detecting changes to memory contents

The DTT model spawns 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 \texttt{tstore()} functions to all assignments that may store a new value to data triggers. These functions replace the conventional store instructions in those cases.

The \texttt{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 \texttt{tstore()} function so that the value is stored properly, and so that changes are detected at the right granularity.

When the \texttt{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 \texttt{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 5.3.2) find that this strategy significantly degrades the performance due to the overhead of spawning new threads. Therefore, we develop a fast thread spawn (Section 5.3.3) 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 dtt_return() call. The dtt_return() 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 dtt_return() function, we also provide a dtt_cancel() function that terminates the support thread when the support thread executes a path that may lead to unwanted results. The dtt_cancel() 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 im-
portant 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 `dtt_barrier()` function right before each skippable region to perform the implicit barrier. The `dtt_barrier()` 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 `dtt_barrier()` function will stall the main thread until all support threads finish execution. If there are no pending support threads associated with the skippable region, the `dtt_barrier()` 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 `dtt_barrier()`, 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.

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
Table 5.1. The three processors have very different microarchitectures that lead to different performance gain and optimal points for software DTT

<table>
<thead>
<tr>
<th>System Information</th>
<th>Intel Nehalem</th>
<th>AMD Opteron</th>
<th>Intel Core 2 Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Model</td>
<td>Nehalem</td>
<td>Opteron 2427</td>
<td>Harpertown</td>
</tr>
<tr>
<td>No of Sockets ×</td>
<td>2×1×4</td>
<td>2×1×6</td>
<td>2×2×2</td>
</tr>
<tr>
<td>No of Dies × No of cores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Cache size</td>
<td>32KB</td>
<td>64KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L1 hit time</td>
<td>4 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L2 Cache size</td>
<td>256KB private</td>
<td>512KB private</td>
<td>6MB (per die)</td>
</tr>
<tr>
<td>L2 hit time</td>
<td>10 cycles</td>
<td>15 cycles</td>
<td>15 cycles</td>
</tr>
<tr>
<td>L3 Cache size</td>
<td>8MB shared</td>
<td>6MB shared</td>
<td>None</td>
</tr>
<tr>
<td>L3 hit time</td>
<td>38 cycles</td>
<td>36 cycles</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>200-240 cycles</td>
<td>230-260 cycles</td>
<td>300-350 cycles</td>
</tr>
</tbody>
</table>

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.

5.2 Experimental 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 previous chapter, we focus on modifications to existing, mature code rather than new programs, which enables better comparison between DTT and non-DTT code.

5.2.1 Processors

To investigate the performance of the software DTT, 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. Table 5.1 lists the configuration of each machine. 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) [53] 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.

5.2.2 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. For each benchmark, we use Pin [44] to profile the memory instructions that frequently incur redundant loads. Unlike the previous chapter that only profiles the portions of execution selected by simpoint [45], 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 5.2 lists our modifications to the benchmarks. These modifications in some cases are identical or similar to those made in the prior work [49], but in several cases (equake, gcc, gzip, mesa, perlbench, and vpr) we target other sections of code because we found that the code used in Chapter 4 did not address whole-program execution as much as it impacted the short simulated regions in the prior experiments. For the same reason, unfortunately, we cannot directly compare the software DTT results to the hardware-support results in the previous chapter, because
Table 5.2. Modifications to selected benchmarks

<table>
<thead>
<tr>
<th>SPEC2000 benchmarks</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>App.</strong></td>
<td><strong>Data triggers</strong></td>
</tr>
<tr>
<td>ammp</td>
<td>last</td>
</tr>
<tr>
<td>art</td>
<td>f1.layer[].P</td>
</tr>
<tr>
<td>bzip2</td>
<td>ss</td>
</tr>
<tr>
<td>crafty</td>
<td>search</td>
</tr>
<tr>
<td>eon</td>
<td>a_MR and a_VHR</td>
</tr>
<tr>
<td>equake</td>
<td>time</td>
</tr>
<tr>
<td>gcc</td>
<td>basic_block.live_at.end</td>
</tr>
<tr>
<td>gzip</td>
<td>max_chain_length</td>
</tr>
<tr>
<td>mcf</td>
<td>node_t</td>
</tr>
<tr>
<td>mesa</td>
<td>img-&gt;Format</td>
</tr>
<tr>
<td>parser</td>
<td>randtable and inputs of count()</td>
</tr>
<tr>
<td>perlbmk</td>
<td>inputs of gv_stashpvn()</td>
</tr>
<tr>
<td>twolf</td>
<td>new_total of dimptr</td>
</tr>
<tr>
<td>vortex</td>
<td>EmpTkns010 and PersonTkns</td>
</tr>
<tr>
<td>vpr</td>
<td>rr_node</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PARSEC benchmarks</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>App.</strong></td>
<td><strong>Data triggers</strong></td>
</tr>
<tr>
<td>blackscholes</td>
<td>data</td>
</tr>
<tr>
<td>bodytrack</td>
<td>pf</td>
</tr>
<tr>
<td>canneal</td>
<td>a and b in annealer_thread::Run()</td>
</tr>
<tr>
<td>facesim</td>
<td>threading_auxiliary_structures-&gt;extended_tetrahedrons-&gt;m</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>cell.a</td>
</tr>
<tr>
<td>swaptions</td>
<td>swaption</td>
</tr>
<tr>
<td>vips</td>
<td>t[12]</td>
</tr>
<tr>
<td>x264</td>
<td>h-&gt;fenc-&gt;i_type</td>
</tr>
</tbody>
</table>
the results are not gathered over identical program regions.

We also investigate the interaction between DTT and traditional parallelism for the first time in this research. We use the PARSEC 2.1 benchmarks [9]. 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 did not overlap 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.

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 blackscholes and swaptions, some of that redundancy is benchmark-related, rather than inherent to the original programs they are based on. However, these results are useful here because the source of the redundancy is less important than whether we can identify it and exploit it.

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 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.

In Chapter A, we will provide detailed descriptions about our benchmark implementation used in this chapter, including the PARSEC benchmark applications.
5.3 Results

This section first quantifies the software overheads of our data-triggered threads implementation, and then 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 hardware 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 [11]. 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 5.2) 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 parallel applications. In our experiments, we use single-thread unmodified benchmarks as the default baseline unless otherwise specified.

### 5.3.1 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 5.1 presents the overhead of our runtime system. 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
Figure 5.2. The relative performance of the software DTTs implementation with both the main thread and the support thread functions running in a single thread.

Figure 5.3. The relative performance of the base software DTTs implementation with the support threads running on separate cores.

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.

5.3.2 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 Chapter 4, we can separate these effects by running both a single-thread version of our DTT run-
time 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 \texttt{tstore()} function detects a memory modification, the program invokes the support thread function immediately in the same hardware context.

Figure 5.2 shows the data-triggered threads performance for a single-thread implementation compared with running unmodified (no DTTs) code on a single core of the processors. 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, equake, mcf, blackscholes, canneal, fluidanimate, and
swaptions, which obtain the most speedup in this configuration, the DTT 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. As in Chapter 4, 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.

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 in Chapter 4.

To benefit from parallelism, we extend our data-triggered threads framework to utilize two cores within a processor, one for the main thread, the other for the support threads. In the baseline multi-threaded DTT framework, the $\texttt{tstore()}$ function creates a new pthread when the system finds a support thread event is available for execution. The $\texttt{dtt\_return()}$ function terminates the thread.

Figure 5.3 shows the performance of our initial software implementation of multi-core DTT. 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 128X performance improvement for swaptions, and marginally better performance for facesim and vips. However, the others all drop off considerably, including blackscholes which achieved 15X 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.

5.3.3 Fast thread spawning

While the initial, naive implementation of software DTT 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, 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 \texttt{dtt\_return()} function updates the state variable of the corresponding skippable region and goes back to the polling function.

Figure 5.4 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 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 DTT implementation. For SPEC2000 benchmarks, the fast spawn scheme improves the performance over
Figure 5.4. The relative performance of the software DTTs implementation with the fast thread spawning mechanism (DTT + fast spawn).
Figure 5.5. The relative execution time of DTT + fast spawn on the Nehalem machine.

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 5.5. It presents the relative portion of time that each benchmark spends 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.

5.3.4 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 (i.e., are redundant), 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 \texttt{tstore()} function will only invalidate the state variable without enqueuing 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 5.6. The relative performance of the software DTTs implementation with fast thread spawning and thresholding mechanism (DTT + fast spawn + threshold), for SPEC.
Figure 5.6 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 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 5.3.6.
Figure 5.7. 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).

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 chapter.

5.3.5 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 5.7 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 5.7. 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).

5.3.6 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, does DTT parallelism overlap with traditional parallelism (exploiting the same phenomena) or is it complementary? Second, how do DTT/parallel programs scale with thread count?

Figure 5.8 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
Figure 5.8. 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.
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 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, the DTT model enables 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.
5.4 Discussion

This chapter presents a pure software approach to support the DTT programming and execution model, software DTT. Software DTT improves the generality and portability of the DTT 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.

Acknowledgements

This chapter contains material from “Software data-Triggered threads” by Hung-Wei Tseng and Dean M. Tullsen, which appears in ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012). The dissertation author was the first investigator and author of this paper. This material is copyright ©2012 by the Association for Computing Machinery (ACM).
Chapter 6

Compiler-generated data-triggered threads

The data-triggered threads (DTT) programming and execution model has shown the ability to improve application performance by exploiting parallelism and eliminating redundant computation. Instead of generating threads based on control flow like traditional execution models, the DTT model spawns a thread when specific memory contents change. The dataflow-like thread generation brings two primary advantages to the DTT model. First, computation that depends on the changed data can execute in parallel immediately. Second, computation that depends on untouched or unchanged data does not need to execute (as a prior execution is still valid). The previous chapters show that the effect of eliminating unnecessary, dynamically redundant computation can be especially powerful.

The DTT model described in the previous chapters relies on programmers’ efforts to achieve performance improvement. That approach needs the programmer to identify the pieces of source code that contain potential for DTT support threads, write the support thread functions, and then attach those functions to variables or structure fields that trigger the computation of the support threads. They do so using extensions to an imperative programming language. In this way, code must be modified to exploit
data-triggered threads or written from scratch. The advantages of DTT remain unavailable for code not written in this style, including existing code that pre-dates the DTT programming model.

In this chapter, we present a compiler framework, CDTT (compiler-generated data-triggered threads), which automatically generates data-triggered threads from applications written in C/C++, without requiring any modification to the source code. Because a support thread function can execute multiple times (if the input changes) before it is consumed, the previous work requires that support thread functions be idempotent. This remains true in our compiler-based approach. The design of CDTT starts by identifying idempotent code regions [19].

Since DTT achieves significant performance improvement through eliminating redundant computation, we also present profile-assisted CDTT, which can use profile-generated information about redundancy as an input to guide the selection of DTT code regions. Profile-assisted CDTT thus selects code regions that contain often-redundant computation and constructs support threads from that code. This work shows that while profile data can be helpful, CDTT is very effective even with no profile input.

In this form, then, CDTT becomes a static compiler optimization that is capable of removing (sometimes large) blocks of code dynamically when that code is found to be redundant. Because it does not require the storage and comparison of inputs that other techniques require (e.g., memoization [37, 14]), it typically does so with relatively low overhead.

The elimination of redundant or unnecessary computation by the compiler can be a highly effective optimization, particularly in modern systems, because it both reduces execution latency as well as power and energy use. Compilers traditionally eliminate \textit{statically redundant} code with techniques such as dead code elimination. Techniques to remove \textit{dynamically redundant} code have been proposed, (e.g., automatic memoiza-
tion [38, 42]), but the cost of checking redundancy of the inputs scales with the size of the inputs themselves. This chapter presents new compiler techniques to remove dynamically redundant code, with costs that are independent of the data structure sizes, allowing code and data regions of unlimited size to be easily exploited.

CDTT can work with either hardware support for DTT in Chapter 4 or the software-only implementation in Chapter 5. To maximize generality, in this chapter we demonstrate CDTT running on top of the software infrastructure on existing hardware. To evaluate the performance of CDTT, we implement the proposed profiling and compilation methods using LLVM [31]. The compiled applications use a DTT runtime system similar to the software DTT infrastructure described in [50]. We achieve 11% average performance improvement over serial SPEC2000 benchmarks with profile-assisted CDTT. With the help of a thresholding mechanism in the runtime system which has the ability to dynamically disable DTT for particular triggers, CDTT without profile data can still achieve a 10% average performance gain.

The success of the non-profiling approach comes from the fact that our algorithm for identifying DTT regions has a high tendency to select code that is redundant.

In this chapter, we will demonstrate that the automatically generated DTTs can achieve nearly the same level of performance improvement as careful human coding. We will describe an analysis framework that identifies potential code regions for applying data-triggered threads. We will also present algorithms for automatically composing support thread functions. Finally, we will demonstrate that the algorithms for selecting regions for DTT formation also serve as an accurate static predictor of redundant computation.
6.1 Design of CDTT

This section describes the CDTT compiler framework. Creating data triggered threads requires the following steps: (1) identifying potential DTT Regions in the existing code, (2) composing support thread functions and skippable regions from each candidate DTT region, and (3) inserting tstores at each data trigger and generating the code for the DTT runtime system. We will discuss each step in turn.

We will initially discuss the algorithms for the case where there is no profiler-generated input. We will then describe the additional steps that would be required when the profile data (about data redundance) is available.

Our compiler framework is built on top of LLVM 3.0 [31]. CDTT accepts C/C++ source code and compiles the program into LLVM bitcode. The LLVM bitcode has a one-to-one mapping to the LLVM intermediate representation. After transforming the LLVM bitcode with DTT runtime support, LLVM generates the x86 machine code.

6.1.1 Identifying potential DTT Regions

The first step of the CDTT framework is to find regions in the original program code that could be transformed into support threads. We will call these DTT Regions. Once selected, a DTT region will serve as a template to create both the support thread and the associated skippable region.

Selecting DTT regions involves the following steps. Throughout this process we maintain a list of candidate regions until the final set of regions are selected.

(1) Identify idempotent regions. Create a list of single-entry, single-exit regions that are idempotent. These regions may overlap with other regions in the list.

(2) Test for name dependence. Remove from the list any region with a possible WAR or WAW data race with surrounding code.
int a_number_DTT()
{
    ATOM *ap;
    if( atomUPDATE )
    {
        atomNUMBER = 0;
        if( first == NULL ) return 0;
        ap = first;
        while(1)
        {
            if( ap->next == NULL)
                break;
            atomNUMBER++;
            if( ap->next == ap )
                break;
            ap = ap->next;
        }
        return atomNUMBER;
    }
}

Figure 6.1. An example of idempotent code

(3) Select DTT regions. Select the largest regions and remove from the list any region that overlaps the selected regions.

Identify idempotent regions  In the beginning of the optimization process, CDTT scans the whole program and creates a candidate list of regions that are single-entry, single-exit code. These regions usually contain multiple blocks, but can be as small as a single block, or as large as a function. These regions may also contain loops.

DTT generates a support thread event when a specific memory content changes. The application can potentially trigger the same code several times before the application uses the result. Therefore, CDTT only creates support thread functions and skippable regions that are idempotent, which means the effect of executing the code region multiple times is the same as executing the code region only once. For the DTT programming model in the previous chapters, it is up to the programmer to create idempotent code. For CDTT, we need the ability to detect idempotence automatically.

CDTT then tests each region in the candidate list for idempotence. Any region that fails this test will be removed from the list. To accomplish the idempotence check,
CDTT needs to identify all the inputs and outputs. A variable is an input of the candidate code region if (1) the variable is used in the region but defined outside of the region or (2) the variable is the source of a memory load instruction within the region. CDTT also identifies the outputs of the candidate code region at this stage. A variable is an output of the candidate code region if (1) the variable is defined in the candidate code region but used outside of the candidate code region, or (2) the variable is the target of a memory store instruction in the candidate code region and the variable is used outside of the candidate code region.

To be idempotent, a code region cannot overwrite any of its own inputs. If the code region overwrites its inputs, the next execution of the code region may use the changed input value and produce a different result. In other words, an idempotent code region cannot contain any data dependency that is a write to an input of the code region. For example, the function in Figure 6.1 is idempotent because the function does not modify any of its inputs. However, if the function code were to assign new values to the `atomUPDATE` variable or if it were to read `atomNUMBER` from the last execution prior to updating, the code would not be idempotent.

To identify potential idempotent regions that the compiler can compose as support thread functions, CDTT uses an algorithm similar to the static analysis phase of Kruijf et al. [19]. CDTT transforms the application code into an LLVM intermediate representation (IR) that is already in SSA form. The SSA form of LLVM IR can eliminate the artificial antidependences that do not affect the idempotence of the examined regions. For each candidate single-entry, single-exit code region, CDTT checks if any data dependency exists – that is a write to the register or memory inputs to the code region. To be conservative, CDTT treats any write to a possible alias (may-alias, partial-alias, or must-alias) of an input to the code region to be a violation of idempotence. If the code region contains no such dependencies, CDTT considers the region as idempotent.
and a potential candidate to be transformed into a support thread function.

Currently, our compiler only detects idempotence – it does not transform the code to try and create idempotence, which would increase the potential for performance gain.

If the candidate code region contains function calls, CDTT also considers the inputs and outputs of calling functions. If the output set intersects with the input set, the candidate code region will not pass our idempotence check. If the code region calls an external function where the function body is unavailable to CDTT, CDTT will not consider this region as idempotent.

**Test for name dependence**  We must not create any data races in our code as we transform serial code into parallel via our support threads. RAW dependence is naturally handled in the placement of triggers and skippable regions, but we must also monitor and prevent name dependences (WAR and WAW). In this step we identify (1) the inputs and outputs for each candidate region, (2) all of the stores to those inputs which will then become potential data triggers, and (3) all of the code reachable between the potential data triggers and the region.

In the reachable code, CDTT identifies both types of name dependence: (1) WAR: any load to any of the outputs of the region and (2) WAW: any store to any of the outputs of the region. For WAW dependency, CDTT also examines the code along the path between two invocations of the region in the reachable code analysis since the DTT model can skip the region several times after a support thread function is triggered. If CDTT identifies any of these dependences for a candidate region, CDTT will remove the region from the list. If the reachable code includes any external function where the function body is unavailable to CDTT, CDTT removes the region from the list. If the region contains any recursive function call that violates these name dependencies,
Figure 6.2. An example inter-procedural control flow graph

CDTT also removes the region from the list.

For example, assume that we have a program as shown in Figure 6.2. The application contains three functions A, B, and C. In basic block A3 of function A, there is an instruction calling function C. In basic block A4 of function A, there is an instruction calling function B. The code region that contains basic blocks B4, B5, B6, and B7 of function B is a candidate region. If an instruction in basic block A1 of function A changes an input of the candidate region, DTT can trigger computation of the candidate region (via a support thread) immediately after the instruction in A1 changes the memory content. However, if function C consumes an output of the candidate region, this creates a potential data race, as the generated support thread now runs in parallel with the code in C, and could write the value before C consumes it.

Together with the memory name dependence information, we use the basic alias analysis infrastructure in LLVM. CDTT considers that two memory objects have no data dependency only when the infrastructure responds with no alias. If the alias analysis tool responds with must-alias, may-alias, or partial alias, CDTT will signal the WAR or
Select DTT regions  To minimize the effect of thread execution overhead, CDTT favors longer support thread functions over short ones. So while at this point our list of candidate regions typically contains a large number of possibly overlapping regions, we seek to select the largest possible regions. Therefore, we begin by ordering all regions by static instruction count. For loops, we multiply their static instruction count by 10x to better estimate the dynamic instruction count without the help of profile data. We select the largest to be a DTT region, then remove from the list that region and all regions that overlap (contain a common basic block) with it. We repeat that process until the list is empty.

Our infrastructure currently does not allow support threads that trigger other support threads – all triggers must be in the main thread. As a result, in this step we must also remove any region that would have a trigger inside an already-selected DTT region.

At the end of this step, we have our set of DTT regions that will be passed to the subsequent phases of the compiler to create support threads, mark skippable regions, and insert tstores.

6.1.2 Generating support thread functions and skippable regions

With the methods described in the previous section, CDTT obtains a list of non-overlapping DTT regions that each need to be transformed into a combination of a skippable region and a DTT support thread.

For each DTT region, CDTT first finds all register and memory inputs/outputs of the code region. Because the DTT model does not support passing arguments other than the triggering address of a support thread event, CDTT uses global variables and
spills the register inputs of skippable regions. CDTT also identifies the outputs of the code region. For register outputs that later code will consume, we still allocate global memory space for storing these outputs. Then, CDTT copies the DTT region code into a new function. CDTT adds instructions at the beginning of the support thread function code to load all the required inputs, and at the end of the support thread function to store register outputs to global memory. After CDTT composes the support thread function, the skippable region is created with minor changes to the original DTT region.

If the underlying architecture contains architectural support for the DTT execution model, CDTT simply needs to add instructions at every exit of the support thread function to complete a support thread event, and provide static information needed for the hardware tables at startup.

If the code is targeting a software-only runtime system, CDTT creates a state variable for each skippable region as in Section 5.1.1 to manage the execution of the support thread event. CDTT adds instructions in the beginning and the end of the support thread functions to manage the state variable. In the beginning of the support thread function, CDTT inserts instructions to load the state variable pointer from the TQ of the DTT runtime system and updates the state to \textit{running}. At the end of the support thread function, CDTT marks the state variable as \textit{valid} or \textit{pending} depending on the current status of the TQ.

Because the beginning of a skippable region is an implicit barrier in the DTT model, CDTT inserts a \texttt{dtt\_barrier} function call before the skippable region to make sure that there is no in-flight support thread event associated with the skippable region. Upon exiting the barrier, the code will check the current status of the state variable associated with the skippable region. If the state variable indicates that the application can bypass the execution of the skippable region, the program will jump past it. Otherwise, the application will execute the code in the skippable region.
6.1.3 Inserting tstores

The DTT model triggers support thread execution when selected memory content changes. Therefore, the compiler needs to identify the store instructions associated with data triggers so that they can be replaced with tstores.

For each support thread, CDTT will find all the potential stores that can trigger that thread. CDTT applies the same approach that we use to discover the producers of inputs of a candidate code region in Section 6.1.1. Because CDTT transforms all the inputs of a support thread function into global variables or memory addresses (Section 6.1.2), all producers of the support thread function are store instructions at this phase. CDTT replaces these with tstore function calls (or tstore instructions if we have hardware support) that check and update memory contents. We also use the basic alias analysis infrastructure in LLVM to identify memory dependencies. If the target of a store instruction is a must-alias, may-alias, or partial alias to an input of the support thread function, CDTT will replace the store instructions with tstore function calls or instructions. It should be noted that poor aliasing could inflate the overhead of CDTT by inducing unnecessary tstores and spurious support threads (because the code is idempotent, there is no correctness issue, only performance). However, we did not encounter this issue in any of our programs.

In some cases, a basic block can contain several tstores for the same support thread function. Triggering all these support thread events is often unnecessary and can potentially result in performance slowdown and energy waste. To minimize the overhead, we introduced a tstore_invalidate function in the DTT runtime system. Unlike the conventional tstore function, the tstore_invalidate function compares and updates the memory content but only invalidates the state variable when the function detects a change of memory content. The tstore_invalidate function will then set a
flag in the state variable, so the last \texttt{tstore} function will trigger a support thread event even if the memory content that the last \texttt{tstore} compares remains the same — this is a slight change to the implementation of \texttt{tstore} from Chapter 5. This works when either the triggering addresses are statically the same, or the support thread is independent of the triggering address (the latter is a common case for our statically-generated data-triggered threads; see the \textit{ammp} code from Figure 6.1 as an example). In these cases, CDTT replaces all the \texttt{tstore} function calls with \texttt{tstore\_invalidate} except for the very last one in the basic block. As a result, the basic block can only trigger one event to the same support thread function each time it executes.

### 6.2 Experimental methodology

We evaluate benchmarks that are written in C or C++ from the SPEC2000 benchmark suites. We select the older SPEC2000 benchmarks to be able to compare with the result in Chapter 5. We use the ref dataset and run each application 5 times to measure the performance. We validate the correctness of programs compiled using CDTT by comparing the output with the original program.

For profile-assisted CDTT, we tried profiling with both the test dataset and the train dataset. This demonstrated that our techniques are tolerant of profile quality, as we achieved essentially the same performance results (on the ref dataset) when profiling with each of them. For the results shown in this chapter, we use train for profiling.

We compile each benchmark application into two different binary versions using LLVM — a highly optimized binary without DTT support and a binary with DTT support using the same compilation flags. When compiling these applications, we add one additional polling thread to execute the support thread function by default. Thus, we never use more than two cores. CDTT only adds at most 20 seconds to compile time across all the applications we examined in this chapter. We use a DTT runtime system without the
thresholding mechanism [50] (we call this multi-core runtime system in the later text) as the default DTT runtime system, but also examine performance with thresholding turned on. For all results shown in this work, we evaluate the performance of CDTT with no hardware support and no programmer modifications.

To investigate the performance of the binaries optimized by our compiler framework, we use an Intel Xeon E5520 (Nehalem) processor as the experimental platform. The 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 support thread on a distinct core in this work.

6.3 Results

This section presents the result of applying the CDTT optimizations to our benchmarks. We examine performance with profiling data (at different cutoffs), without profiling data, with and without thresholding to control DTT spawning, and examine the runtime overheads of CDTT.

6.3.1 Performance of profile-assisted CDTT

CDTT can work either with or without profile data. For CDTT with profile data (profile-assisted CDTT), the profile data and the silent store cutoff are two important inputs. To see how profile data and different silent store cutoffs affect the performance of applications, we perform experiments that change the silent store cutoff from 0% to 90% with increments of 10%. We also examine a highly conservative 99% silent store cutoff. When the silent store cutoff is 0%, this is equivalent to CDTT with no profile data.

Table 6.1 shows the number of support thread functions and the average number of static instructions in these support thread functions for each application. We list
According to the table, profile-assisted CDTT can generate more and (in many cases) longer support thread functions when we relax the silent store cutoff constraint or turn off the profile data. All applications identify at least one redundant region, and those regions are often of reasonable static size. Although each program contains a fair number of potential idempotent regions, the actual number of DTT regions is typically small. This is a result of two factors. First, we eliminate a number of candidates due to the name dependence tests, and second, our algorithm tends to coalesce smaller regions into a small number of larger regions.

Figure 6.3 presents the performance of the applications under different silent store cutoffs. We run each application with the default multi-core runtime system. In this graph, we use PACDTT-X to represent profile-assisted CDTT with X% as the silent store cutoff. We use CDTT to represent the case where we use CDTT without any profile data. We skip the 10%, 30%, 50%, 60% and 70% cases because their performance does not dramatically differ from the neighboring data points. On average, CDTT performs best when the silent store cutoff is 80%. We obtain an average 10% speedup across
Table 6.1. The average static instruction counts for the support thread functions for applications compiled using profile-assisted CDTT with silent store cutoffs of 80%, 20% and CDTT without profile (0%)

<table>
<thead>
<tr>
<th>Name</th>
<th>Support thread functions</th>
<th>Average support thread instruction counts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>80%</td>
</tr>
<tr>
<td>ammp</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>art</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>bzip2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>crafty</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>eon</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>equake</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gcc</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>gzip</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>mcf</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mesa</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>parser</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>perlbmk</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>twolf</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>vortex</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>vpr</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

all benchmarks, and the best performance improvement is for mcf, at 55%. CDTT with the 80% silent store cutoff results in a 5% reduction of dynamic instructions over the C SPEC2000 applications, compared to no cutoff. We also observe that CDTT achieves 6% performance improvement even without profile data. We will provide a more detailed discussion of CDTT without profile data in Section 6.3.2.

For gcc and mesa, the silent store cutoff affects the performance significantly. In these cases, lower silent store cutoffs increase both the number of triggers (instances of the tstore function) and the number of executed support threads, potentially resulting in a significant increase in the total dynamic instruction count. For example, with gcc, the dynamic instruction count increases by a hefty 60%. Profile-assisted CDTT performs much better than without using profiling for these benchmarks. The large gains with profiling in gcc indicate that CDTT does identify good DTT regions; however, when the cutoff is too low, the good regions become dominated by the negative overheads of the
Figure 6.4. The speedup of profile-assisted CDTT with different silent store cutoffs running on the runtime system with thresholding.

useless regions. With the profile data, we can filter out the useless regions and isolate the useful regions.

In other cases, too high a silent store cutoff can overly constrain the construction of support threads. For example, profile-assisted CDTT with 99% silent store cutoff creates no support threads for gzip, twolf, and vpr.

For several benchmarks, the silent store cutoffs do not make a significant difference in performance because the regions that pass single-entry, single-exit idempotence requirements already tend to have highly redundant computation. This will also be explored further in the next section.

To prevent frequent stalling of the main thread resulting from ill-behaved support threads, the software DTT [50] paper proposes a thresholding mechanism that dynamically disables the usage of DTT when the main thread frequently has to stall at the barrier, waiting for support threads to complete. Figure 6.4 presents the application performance when profile-assisted CDTT uses the DTT runtime system with thresholding, targeting different silent store cutoffs.

Relative to Chapter 5, we find that thresholding is actually less important for the automatically generated threads using profile-assisted CDTT than for programmer-
Table 6.2. The percentage of silent stores in DTT regions selected by CDTT (without profiling), compared to the percentage of silent stores in all code.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>DTT regions</th>
<th>All code</th>
<th>benchmark</th>
<th>DTT regions</th>
<th>All code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>100%</td>
<td>0.3%</td>
<td>art</td>
<td>40%</td>
<td>47%</td>
</tr>
<tr>
<td>bzip2</td>
<td>100%</td>
<td>4%</td>
<td>crafty</td>
<td>89%</td>
<td>22%</td>
</tr>
<tr>
<td>eon</td>
<td>100%</td>
<td>31%</td>
<td>equake</td>
<td>100%</td>
<td>1%</td>
</tr>
<tr>
<td>gcc</td>
<td>58%</td>
<td>39%</td>
<td>gzip</td>
<td>15%</td>
<td>10%</td>
</tr>
<tr>
<td>mcf</td>
<td>100%</td>
<td>50%</td>
<td>mesa</td>
<td>64%</td>
<td>43%</td>
</tr>
<tr>
<td>parser</td>
<td>81%</td>
<td>10%</td>
<td>perlbmk</td>
<td>91%</td>
<td>11%</td>
</tr>
<tr>
<td>twolf</td>
<td>40%</td>
<td>40%</td>
<td>vortex</td>
<td>100%</td>
<td>57%</td>
</tr>
<tr>
<td>vpr</td>
<td>39%</td>
<td>2%</td>
<td>average</td>
<td>74%</td>
<td>24%</td>
</tr>
</tbody>
</table>

generated threads. This is because the profiling allows us to be fairly conservative and avoid ill-behaved threads that need to be disabled at runtime.

We do find, however, that the optimal silent store cutoff is much lower in the presence of dynamic thresholding – 80% in the absence of thresholding, but 20% when thresholding is turned on. This is expected, as thresholding allows the compiler to be more liberal, with the runtime system still able to eliminate poorly chosen DTT threads. In the best case (20% silent store cutoff), we get an average of 11% performance gain.

### 6.3.2 Discussion of CDTT without profiling

Perhaps most interesting, however, are the results in Figure 6.4 for the no-profile results. First, we see that thresholding is more important for the no-profile results than the profiled results – again, this is expected because the no-profile results apply no compile-time filter and must rely more heavily on the runtime filter.

Second, we observe that the no-profile results, with thresholding, achieve a 10% overall gain, only slightly below the profile-assisted result (and as we’ll see in a following section, competitive with previous hand-coded results). This is not an expected result, that we lose little performance by ignoring extensive information about the redundancy of computation.
Table 6.2 helps illuminate this phenomenon. It describes the percentage of stores that are silent, both for the total application, and for the code that is selected to be placed in data-triggered threads (for the case where our algorithms do not use any profile information). In many cases, we see a dramatic difference in the likelihood of computation being redundant (as indicated by the silent stores) in the selected code vs. the remaining code. It turns out that our algorithms for selecting DTT regions are a highly effective static predictor of redundant code.

Consider a potential DTT region that is composed of a few loads, some computation that depends on those loads, and ends in one or more stores. If all of the loads are redundant, the stores will be silent. Our selection criteria (namely idempotence and name dependence) tend to filter out loads unlikely to be redundant. Idempotent regions are those where the loads do not depend on the region itself. The name dependence analysis filters out regions with loads that depend on the surrounding code (anywhere between the trigger and the skippable region). Thus, a region is only selected if there are no stores to the loaded data anywhere in the nearby, reachable code (either within or outside the DTT region itself). Thus, it is not surprising that the selected loads are much less likely to be written to, and the stores that depend on those loads are far more likely to be silent.

As a result, by selecting only DTT regions with minimal interactions with surrounding code, our DTT region selection criteria also doubles as an effective static predictor of redundancy. Therefore, in many cases, the carefully collected profile data serves only to confirm the identification of the redundant regions.

6.3.3 Redundance vs. parallelism in CDTT

Like prior applications of DTT described in the previous chapters, CDTT exploits both redundancy and parallelism. In this section, we perform experiments that
### Figure 6.5.
The speedup of CDTT on single-threaded and multi-core runtime system

<table>
<thead>
<tr>
<th>Application</th>
<th>Single-threaded CDTT</th>
<th>CDTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>1.06</td>
<td>1.24</td>
</tr>
<tr>
<td>art</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>crafty</td>
<td>0.65</td>
<td>0.89</td>
</tr>
<tr>
<td>eon</td>
<td>1.55</td>
<td>1.12</td>
</tr>
<tr>
<td>equake</td>
<td>1.00</td>
<td>1.06</td>
</tr>
<tr>
<td>gcc</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>gzip</td>
<td>0.99</td>
<td>1.13</td>
</tr>
<tr>
<td>mcf</td>
<td>1.03</td>
<td>1.08</td>
</tr>
<tr>
<td>mesa</td>
<td>0.85</td>
<td>1.10</td>
</tr>
<tr>
<td>parser</td>
<td>1.06</td>
<td>1.11</td>
</tr>
<tr>
<td>perlbench</td>
<td>1.20</td>
<td>2.44</td>
</tr>
<tr>
<td>twolf</td>
<td>127.5</td>
<td></td>
</tr>
<tr>
<td>vortex</td>
<td>1.02</td>
<td>29.4</td>
</tr>
<tr>
<td>vpr</td>
<td>16.3</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.06</td>
<td>1.10</td>
</tr>
</tbody>
</table>

### Figure 6.6.
The speedup of CDTT without profile (CDTT), profile-assisted CDTT with 20% silent store cutoff (profile-assisted CDTT) and programmers’ modification (hand-coded DTT) running on multi-core runtime system with thresholding

<table>
<thead>
<tr>
<th>Application</th>
<th>Profile-assisted CDTT</th>
<th>Hand-coded CDTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>1.07</td>
<td>1.17</td>
</tr>
<tr>
<td>art</td>
<td>1.25</td>
<td>1.15</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.00</td>
<td>1.05</td>
</tr>
<tr>
<td>crafty</td>
<td>1.03</td>
<td>1.12</td>
</tr>
<tr>
<td>eon</td>
<td>1.00</td>
<td>1.12</td>
</tr>
<tr>
<td>equake</td>
<td>1.24</td>
<td>1.08</td>
</tr>
<tr>
<td>gcc</td>
<td>1.06</td>
<td>1.00</td>
</tr>
<tr>
<td>gzip</td>
<td>1.57</td>
<td>1.55</td>
</tr>
<tr>
<td>mcf</td>
<td>1.21</td>
<td>1.17</td>
</tr>
<tr>
<td>mesa</td>
<td>1.06</td>
<td>1.15</td>
</tr>
<tr>
<td>parser</td>
<td>1.04</td>
<td>1.17</td>
</tr>
<tr>
<td>perlbench</td>
<td>1.21</td>
<td>1.12</td>
</tr>
<tr>
<td>twolf</td>
<td>127.5</td>
<td>14.8</td>
</tr>
<tr>
<td>vortex</td>
<td>1.02</td>
<td>29.4</td>
</tr>
<tr>
<td>vpr</td>
<td>16.3</td>
<td>29.6</td>
</tr>
<tr>
<td>Average</td>
<td>1.06</td>
<td>1.11</td>
</tr>
</tbody>
</table>

allow us to separate the effects.

Figure 6.5 compares the performance result of (1) running both the main thread and support thread functions in the same thread as *(single-threaded CDTT)* with (2) CDTT running on two cores of a multi-core runtime system *(CDTT*, with the main thread on one core and another core devoted to executing DTTs). With the single-threaded configuration, the program executes the support thread function immediately after the application modifies a data trigger without using an additional thread. This provides us two insights. First, it allows us to distinguish between the two advantages of DTT,
redundance and parallelism, because the single-threaded version can only exploit the former. Second, it provides an interesting comparison with serial execution, because it only utilizes one core for DTT. For this graph, we only show the no-profile, no thresholding results – while this is not our best result, thresholding is not available for the single-thread implementation, and profiling filters for redundancy and thus masks one of the phenomena (parallelism) this graph is trying to identify.

For most applications, we see only small improvement moving from the single-threaded runtime system to the multi-core runtime system, indicating that the primary gain comes from eliminating redundant computation. This is not surprising as CDTT favors regions with redundancy. For benchmarks like gzip and twolf, we find that the DTT parallelism does help improve performance significantly. We see this because the multi-core implementation of CDTT achieves gains that the single-thread version cannot; in those cases, the overheads of CDTT actually increase instruction count causing performance loss for the single-thread version, but increased parallelism allows the multi-core version to still achieve speedup.

When profile data is incorporated (results not shown), the outcome is predictably different. Because the profile data targets redundancy specifically, the difference between the single-thread and the multi-core results is only 0.4% (when using the 80% cutoff), implying that virtually all of the gains in that case are from removing redundant computation.

The single-thread CDTT results are also important because they maximize the potential energy advantages of this technique – that potential is significant, because the most effective energy optimization is to not do computation, which is the strength of CDTT. However, in the dual-core case the gains are mitigated by the spinning, often idle polling thread. In the best case, for example, mcf running in single-thread mode expends only 65% of its original energy, resulting in an energy-delay product that is
only 42% of its original value – that is a 2.4X gain in energy efficiency. These values were measured at the wall using a power meter.

### 6.3.4 CDTT and hand-coded DTT

Figure 6.6 compares our automatically generated results with the carefully hand-coded results from Chapter 5. We use the multi-core runtime system with thresholding across all the experiments. We actually match the highest gain, mcf, from hand-coding. With profile data, our automatic system optimizes the same code region, the while loop in the refresh potential function. Without profile data, CDTT also optimizes the primal bea mpp function that the previous hand-coded version did not target and provides an additional, but small performance gain.

On the other hand, gcc compiled with CDTT achieves strong gains that the hand-coded versions missed – our framework allows CDTT to create support thread functions for code regions in reload as needed, expand call, and mark set 1. In one case, for example, we have silent stores that depend only on registers (either locals or function arguments). There are no redundant loads in the region for the programmer (led by the original profiles) to use as triggers, but when our system identifies the region the inputs get placed in memory, giving us redundant loads to use as triggers.

For most other benchmarks, CDTT does not select the same regions that the hand-coded version did. This happens for several reasons.

First, in Chapter 5, we composed code using profile data of redundant loads, but CDTT works with profile data of silent stores, or using only static analysis for idempotence. Sometimes this helps us – CDTT gets additional gain on art and vortex by exploiting functions that execute frequently but the original profiling data did not indicate as promising. In art, CDTT optimizes the code regions in the match function which is called more frequently than the train_match function the hand-coded version
focused on because of the high load redundancy. In vortex, CDTT selects blocks in
the DbmGetVchunkTkn function that executes 39x more than the PersonObjs_FindIn
function that the hand-coded version targets.

In addition, CDTT must be conservative with respect to data races and idempotence. However, the programmer can ignore potential data races that do not occur or create idempotent code where there is none. As an example of the former case, in bzip2, CDTT does not select the same region as the hand-coded version in the sortIt function because the code region calls several shared library functions that then fail our name dependence tests. This is also the case for eon and vpr, two of the strong hand-coded performers.

For equake, mesa, and twolf, some important redundant code, even though it is identified by the profiler, does not fall within an idempotent region. In those cases, the programmer can usually write idempotent DTTs, while our system does not currently have the ability to create idempotence. In other cases, the programmer can replace the original computation with an incremental version, which our infrastructure also cannot replicate.

Chapter 5 showed very high speedups for the PARSEC benchmarks (e.g., 16X on blackscholes and 128X on swaptions). In this chapter, we have not shown those results in earlier graphs. This is because some of that redundancy is benchmark-related as described in Section 5.2.2. However, these results are still useful here because we want to show the fact that CDTT can achieve the same level of performance improvement as programmer’s modification. Thus, the PARSEC results are also included in Figure 6.6. Some benchmarks from PARSEC that the prior work used are currently incompatible with our toolchain: canneal contains inline x86 assembly that cannot be converted into LLVM IR and cannot be supported by LLVM JIT; facesim contains C++ exceptions; vips cannot be compiled correctly using LLVM. We run PARSEC benchmarks with sim-
Figure 6.7. The runtime overhead of the runtime system that CDTT uses large dataset to collect profile data and use the native dataset to measure the performance. The experimental result in Figure 6.6 shows that our framework actually identifies the redundant regions and replicates the spectacular hand-coded gains on those PARSEC benchmarks.

On average, despite no programmer involvement whatsoever, we still achieve nearly all of the performance gain achieved with hand-coding, both for the SPEC and the PARSEC results.

6.3.5 Runtime system overheads

In this work, we evaluate the performance of the proposed compiler framework using a runtime system assuming no special hardware support. The runtime must detect trigger-induced changes to memory and manage the support threads.

To evaluate these costs, we implement a runtime system that performs memory change detection and most of the thread management features except that the runtime system does not actually generate the support thread and does not skip any computation. Thus, it experiences almost all of the overhead and none of the benefits. To separate the effect of additional global variables that CDTT adds to store inputs for skippable regions
and communication between the main thread and support threads, we also evaluated
the performance of non-DTT version binaries using these global variables. Figure 6.7
depicts the overhead of our runtime system. We present the performance overhead for
applications compiled with profile-assisted CDTT and CDTT without profile.

When using profile-assisted CDTT, the overhead of our runtime system is 1.4%
on average. The runtime system overhead is low because only a fraction of all stores
are associated with data triggers, and only those stores experience the tstore overhead
(and then possibly generate thread management overheads). Take gcc as an example,
when the silent store cutoff is 80%, only 0.5% of stores are tstores. However, when the
silent store cutoff is less than 20%, the number of tstores increases by 5X and results
in significant overhead in comparing the values and managing the threads. For bench-
marks where CDTT inserts many tstore or tstore Invalidate calls, like gcc, the
performance degradation can be large. For example, gcc suffers a 44% performance
degradation. The additional global variables added by CDTT only incurs another 0.2%
of performance degradation because profile-assisted CDTT only creates a limited num-
er of skippable regions and support thread functions within the code.

In the absence of profile data (CDTT), the compiler is much more liberal in
generating support thread functions, resulting in more stores being identified as data
triggers, as well. In that case, the runtime system overhead results in an average 3.2%
performance degradation for SPEC2k benchmarks. The additional global variables of
CDTT contribute to another 1% of the performance degradation because CDTT without
profile data also creates more skippable regions and support thread functions.

These results do give us some insight into the possible performance of a system
with hardware support for DTT as in Chapter 4. In that case, nearly all of the measured
overhead will go away. This implies that our non-profiled results could improve signifi-
cantly (from the current 11% to over 14%), as the 3.2% overhead will disappear. These
results also indicate, however, that when the compiler has profile data and is configured to be fairly conservative, the expected gain from hardware support is relatively small and the software system appears to be sufficient.

6.4 Discussion

This chapter presents a compiler framework which can automatically generate binaries which identify dynamically redundant code and bypass the redundant computation. It generates data-triggered thread executables from existing conventional source code. The CDTT binary runs on top of a software runtime system. The compiler framework allows a set of serial applications from SPEC2000 to be sped up by 11% on average for the SPEC benchmarks (as high as 57%), without any code modification and no hardware support. The result for the PARSEC benchmarks is even higher. Energy efficiency gains are even greater, since most of the performance gains come from not doing work.

A key insight of CDTT is that idempotence and name dependence analysis becomes a highly effective static filter for redundant code identification, rendering profiling unnecessary.

Acknowledgements

This chapter contains material from “CDTT: Compiler-generated data-triggered threads” by Hung-Wei Tseng and Dean M. Tullsen, which appears in the 20th International Symposium on High Performance Computer Architecture (HPCA 2014). The dissertation author was the first investigator and author of this paper. This material is copyright ©2014 by the Institute of Electrical and Electronics Engineers (IEEE).
Chapter 7

Related work

This chapter places the DTT model in context relative to the previous execution models and architectures that triggers computation upon the generation of data. This chapter also describes several programming models and compiler techniques that can eliminate redundant computation or opens new opportunities for parallelism as the DTT model.

The DTT model can find its roots in dataflow architectures. In contrast to the von Neumann model that executes instructions using the program counter order, instruction scheduling in dataflow models[20, 4, 43, 5] is only based on operand availability, and execution is not constrained by program sequencing. In this way, instructions with their operands available can be processed concurrently to achieve fine-grain parallelism. The DTT model triggers parallelism upon the generation of data, like dataflow architectures to open new opportunities for parallelism.

The dataflow-like execution model brings another advantage to the DTT model – the computer can avoid unnecessary computation when data remain the same. Several hardware approaches that use hardware tables to cache input-output pairs for instructions, blocks, or functions can also eliminate redundant computation [47, 25, 33, 32]. The programmer can also use programming techniques like memoization [37, 14] to reduce redundant computation.
Similar to Cilk [22] and CEAL [24], the DTT model incorporate the concept of dataflow programming models into imperative programming languages. Cilk targets dataflow-like parallelism, and CEAL focuses on incremental recomputation to avoid redundant computation. The DTT model allows the programmer to benefit from both parallelism and incremental recomputation without completely rewriting programs using pure dataflow programming models [13, 40].

CDTT [51] creates parallelism that initiates using the DTT model from legacy programs similar to Program Demultiplexing (PD) [8]. There are also existing compiler optimizations that can eliminate redundant computation as CDTT [10, 29].

In the following paragraphs, we will describe these works in detail.

7.1 The dataflow model

The basic dataflow execution model [20] represents a program as a collection of operations describing the dataflow graph. To extract the full power of dataflow execution model, the programmer uses pure dataflow programming languages [13, 40] that differ significantly from imperative programming languages to describe programs as dataflow graphs. The processor using the dataflow model triggers instructions only based on operand availability. A completed instruction directly forwards the result to the target instructions without saving any state, so the execution model is stateless and does not incur any overhead when context switching. The synchronization of parallel instructions is implicit by direct result forwarding. This initial implementation of the dataflow architecture can only provide limited parallelism but cannot support general recursion. Therefore, dynamic dataflow architectures [4, 43, 5] use tokens that attach a tag to each value to improve the generality of the dataflow architecture. An operation can execute only when the operation receives all input tokens with the same tag number. However, these classic dataflow processors are hampered by the hardware complexity
for communication and token matching.

To provide a smoother transition from the von Neumann model to dataflow architectures, hybrid architectures\[27, 39, 17\], StarT[41, 3], EARTH[26], DDM[21, 30], and Fuce[2] attempt to build dataflow machines on top of conventional architectures. Similar to the DTT model, these hybrid architectures exploit fine-grained (instruction level) parallelism using the von Neumann model based processors. In terms of coarse-grained parallelism (thread level), these machines adopt the dataflow model to schedule threads. This allows hybrid architectures to support imperative programming interfaces and existing instruction set architectures. However, existing proposals still require significant changes to a baseline architecture to support message passing, thread management, synchronization, context switching, and memory accesses.

### 7.2 Eliminating redundant computation

A number of prior proposals have exploited redundant computation. These works usually employ hardware tables to cache the inputs and outputs from previous execution instances. The processor can use the cached outputs in the future computation instead of performing the same computation again if the inputs remain the same \[47, 25\]. For example, dynamic instruction reuse \[47\] buffers the inputs and execution results of instructions. The processor can skip the execution stage of reused instructions if the inputs match saved prior invocations. Thus, each instruction is reused non-speculatively as long as the inputs match. Block reuse \[25\] expands the reuse granularity to a basic block by tracking and storing inputs and outputs of each block. Similarly, if the processor detects the same set of inputs to the basic block, the processor can skip the execution of the basic block. However, because the usage of hardware tables, these techniques are significantly limited in the size of the computational blocks that they can reasonably address and all but the last are also very limited in the number of addresses they can
track. Moreover, these techniques do not work if a single instruction or code block has multiple frequently used input values.

Memoization [37, 14] is a technique, typically employed in software, that stores the input and output values of frequent operations or functions. When the input values repeat, the program can reuse the output to avoid recalculation. Memoization requires the programmer to create additional storage in software to hold the values of all inputs. Sometimes, the programmer needs to significantly change the algorithms to apply memoization. Conversely, the DTT model triggers proactively as soon as value(s) change without the need to check sameness before skipping redundant computation. Therefore, DTT works with almost no storage, works for code regions of any size, allows unlimited data structure sizes, and naturally exposes and exploits parallelism. Because of the storage limitations, in particular, only a small fraction of the software changes we exploit with DTT could be reasonably reproduced with memoization.

Silent stores[33, 32] detects and removes store instructions from the load/store queue by exploiting the free read ports of the cache. The architectural support of the DTT model (Chapter 4) leverages the silent store detection logic from these researches to compare the writing value and the existing value. In addition to removing the silent store instruction, the DTT model has the potential to remove the whole computation string leading to the silent store.

### 7.3 Dataflow-like programming models

Cilk [22] and CEAL [24] incorporate the concept of dataflow programming models into imperative programming languages to take advantage of dataflow constructs without completely rewriting programs. Like the DTT model, they each propose extensions to the C/C++ programming language to trigger computation when the program generates new data.
Similar to the concept of support thread functions in the DTT model, each thread in Cilk is an instance of a non-blocking C function. The Cilk programming model uses the return values and arguments to these non-blocking functions to specify the data dependencies among threads. The Cilk runtime system does not execute a thread until the runtime system receives all the arguments for the non-blocking function. DTT shares the idea of triggering parallel non-blocking threads upon the generation of data with Cilk, but Cilk does not exploit the potential of removing redundant computation.

CEAL and the DTT model both provide general-purpose language extensions to allow incremental recomputation. The programmer implements a core and mutators for each program using C language extensions. The mutators reflect the change of data and propagate the computation results to the core program. In this way, both CEAL and the DTT model only operate on changing data and avoid redundant computation on unchanging data. However, CEAL does not incorporate parallelism or threading into its solution.

7.4 Compiler optimizations for eliminating redundant computation and creating parallelism

Several compiler optimizations [10, 29] target redundant computation. Bodik et al. [10] removes the redundant computation along frequently executed paths and speculatively executes the optimized code. The IA-64 compiler utilizes the additional registers in the processor to create temporary storage for data that are highly likely to be accessed in the near future. Therefore, the IA-64 compiler can reduce the amount of memory accesses. However, these compiler optimizations only work on relatively small blocks of code, and on load instructions with a relatively small memory footprint.

Both CDTT and Program Demultiplexing (PD) [8] try to create non-traditional concurrency from programs written in imperative languages. PD decouples the exe-
cution of methods from the total sequential ordering of the program. Like the data triggers in CDTT, PD can insert triggers to the sites where the inputs of each demultiplexed method change. The program can then initiate the execution of a demultiplexed method as soon as the evaluation result of a trigger becomes true. Because PD executes these demultiplexed methods speculatively, PD requires additional hardware to buffer the speculative results. In addition, PD never decreases the instruction count because it does not have the ability to skip redundant computation.

Like CDTT, de Kruijf et al. [19, 18] use idempotent code. These works take advantage of idempotence to support low-overhead fault recovery. They identify the idempotent code regions in the program. When the system detects an error within the idempotent region, the system uses the idempotent code to reconstruct the program states since re-executing idempotent code does not affect the correctness of the program. In this way, the system does not need to create checkpoints for idempotent code regions. In addition to idempotence analysis, the CDTT compiler detects possible name dependencies and other filters that are not necessary in de Kruijf et al’s work.
Chapter 8

Conclusion and future work

It is critical to the computer industry, both hardware and software, that we continue to scale both the raw performance and energy efficiency of applications. We will need to exploit architectures, programming languages, compilers, and new programming models to achieve this. This thesis, the data-triggered threads model, in particular, presents a new programming and execution model which makes it natural to express computation in such a way that redundant computation is eliminated, and exposes new opportunities for parallel execution.

In particular, the DTT model allows the programmer to express computation that only executes upon modifications to data. We show that 78% of the loads in the C SPEC benchmarks are redundant and create unnecessary computation. By making small changes to hardware and existing C programs to exploit the data-triggered thread execution model, we achieve speedups as high as 5.89, and averaging 1.46. This work also demonstrated that even without any hardware support, the software DTT can still speed up the same set of applications by 15%. The complete set of serial applications (including single-thread PARSEC) were sped up by 7.3X (arithmetic mean) or 1.6X (geometric mean). The DTT model can be highly complementary with traditional parallelism and achieve significant performance gain, as high as 64X, even over the original parallel version.
This work also presents CDTT, a compiler framework that takes C/C++ code and automatically generates a binary that eliminates dynamically redundant code without programmer intervention. Using idempotence analysis and inter-procedural name dependence analysis, CDTT identifies potential code regions and composes support thread functions that execute as soon as live-in data changes. The compiled binary running on top of a software runtime system can achieve nearly the same level of performance as careful hand-coded modifications in most benchmarks. CDTT improves the performance of serial C SPEC benchmarks by as much as 57% (average 11%) on a Nehalem processor.

The DTT model has the potential to yield orders of magnitude improvements in energy-delay product. This is because in some cases we achieve multiplicative increases in performance, and that increase comes not from doing the same work in less time, but by doing less work, yielding a similar decrease in energy. The most effective way to save energy is to do less work. Much good architecture research has focused on doing less work per instruction. We take an alternate approach – in some cases, we simply skip blocks of several million instructions.

The growing number of computing devices, social networking applications, online services, and online business transactions leads us to an era of data explosion. As of 2012, the world created an average of 2.5 exabytes of new data every day [1]. Data-centric computing – which processes data in a data-oriented approach – is an alternative for applications with huge amount of data. However, this is not a natural transformation for convention parallel programming models, which create parallelism by partitioning the computation. The DTT model always attaches computation to data. The current DTT runtime system or hardware only provide limited support in massive parallelism. In the future, we will exploit the potential of attaching computation to data in the DTT model to provide a platform for big data applications.
Appendix A

Benchmark implementations

In this chapter, we detail the implementation for each benchmark we used in this thesis. Section A.1 - Section A.15 describe our modifications of SPEC2000 benchmarks. Section A.16 - Section A.23 present our changes in the PARSEC benchmarks.

For each benchmark, we first give a high level description of our modifications and then summarize the code sections we modified.

A.1 ammp

The ammp benchmark is a computational chemistry application that models molecular dynamics. Our profiler indicates one of the most time-consuming function, the `a_number` function, contains more than 99% redundant loads. Therefore, we apply the DTT model to this function.

The `a_number()` function reads the global variable `atomUPDATE` and counts the total number of atoms in the linked list. The `atomUPDATE` variable is set whenever the application changes a field of an element in the list of the `ATOM` data structure. However, unless the application adds a new `ATOM` into the list, which modifies the `next` field in any of the elements in the list, the total number of elements in the list remains the same. As a result, the original code repeatedly reads the same values from `next` pointers and returns the same `atomNUMBER`. 

100
Since we only need to update the value of atomNUMBER when a next pointer changes, we attach the data trigger to the next field. The DTT model will initiate the support thread function, a_number_DTT() when the application modifies any next field in the ATOM data structure. This support thread function walks through the linked list to calculate the up-to-date atomNUMBER. Once a_number_DTT() updates atomNUMBER, the application does not need to recalculate the number of ATOMS in the a_number function. Therefore, we mark the region of code that performs this part of computation in the a_number function as the skippable region.

The following code summarizes our modifications:

**Data trigger declaration**

```
typedef struct{
    float x,y,z;
    critical_precision fx,fy,fz;
    int serial;
    void *next; #trigger a_number_DTT()
    ....
} ATOM;
```

**The support thread function**

```
#DTT a_number
int a_number_DTT() {
    ATOM *ap;
    atomNUMBER = 0;
    if( first == NULL ) return 0 ;
    ap = first;
    while(1) {
        if( ap->next == NULL)
            break;
        atomNUMBER++;
        if( ap->next == ap )
            break;
        ap = ap->next;
    }
    return atomNUMBER;
```
A.2 Art

ART implements a neural-network based algorithm to recognize objects in a thermal image. In the SimPoint-based result used in chapter 4, we found that `train_match` is the most critical function. Our profiler in the Software DTT work of Chapter 5 shows that the `match` function is the most critical one. Though the profiling tools identify different functions, the behavior of both functions are similar. They both contain more than 99% redundant loads when loading P values from the f1_neuron array, f1_layer. Both functions contain code regions computing the Y values and finding match that only depends on the P values.

Therefore, we apply the DTT model by attaching a data trigger to the P field in
the data structure and perform the computation that only depends on the P values. In Chapter 4, we apply this modification to the train_match function, and we apply this modification to the match function in Chapter 5.

However, because we need to keep the support thread function idempotent, the support thread function cannot be incremental in this case. We always recalculate the y values in the Y array in the above code, and as a result, we can potentially increase the instruction count if the redundancy of P values is not high enough.

The following code sections demonstrate the data trigger declaration, the support thread function and the skippable region that we use in Chapter 5. For Chapter 4, because the train_match function contains the same computation as in match, we use the same data trigger as in Chapter 5, but replace the computation in the train_match function.

**Data trigger declaration**

```c
typedef struct {
    double *I;
    double W;
    double X;
    double V;
    double U;
    double P;  #trigger match_compute_winner_thread();
    double Q;
    double R;
} f1_neuron;
```

**The support thread function**

```c
#DTT computeWinner
void match_compute_winner_thread(f1_neuron *x) {
    int ti, tj;
    for (tj=0;tj<numf2s;tj++) {
        Y[tj].y = 0;
        if ( !Y[tj].reset )
            for (ti=0;ti<numf1s;ti++)
```
Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}

/* Find match */
*Winner = 0;
for (ti=0;ti<numf2s;ti++) {
  if (Y[ti].y > Y[*Winner].y)
    *Winner =ti;
}
return;
}

The skippable region

for (tj=0;tj<numf1s;tj++)
  f1_layer[tj].Q = f1_layer[tj].P;
#block computeWinner
  /* Compute F2 - y values */
  for (tj=0;tj<numf2s;tj++)
  {
    Y[tj].y = 0;
    if ( !Y[tj].reset )
      for (ti=0;ti<numf1s;ti++)
        Y[tj].y += f1_layer[ti].P * bus[ti][tj];
  }
/* Find match */
winner = 0;
for (ti=0;ti<numf2s;ti++)
{
  if (Y[ti].y > Y[winner].y)
    winner =ti;
}
#end_block

A.3 bzip2

The bzip2 benchmark in SPEC2000 implements the core algorithm of bzip2-0.1 and performs all compression and decompression in memory. Our profiler found that more than 99% of the loads from the ftab table in the most frequently executed sortIt
function are redundant. In each iteration, the \( ss \) variable decides the entry the iteration will access in the iteration, so we attach the data trigger to this variable. In this way, we only trigger the support thread function, the \texttt{bbstart\_thread} function, to update values depending on the \texttt{ftab} table when \( ss \) changes. We also make the original code in the \texttt{sortIt} function the skippable region.

We illustrate our modifications below:

\textbf{Data trigger declaration}

\begin{verbatim}
Int32 ss; #trigger bbstart_thread();
\end{verbatim}

\textbf{The support thread function}

\begin{verbatim}
#DTT bbstart
void bbstart_thread(int *x) {
  Int32 ss = *x;
  bbStart = ftab[ss << 8] & CLEARMASK;
  bbSize = (ftab[(ss+1) << 8] & CLEARMASK) - bbStart;
  shifts = 0;

  while ((bbSize >> shifts) > 65534) shifts++;
  return;
}
\end{verbatim}

\textbf{The skippable region}

\begin{verbatim}
if (i < 255) {
  #block bbstart
    Int32 bbStart = ftab[ss << 8] & CLEARMASK;
    Int32 bbSize = (ftab[(ss+1) << 8] & CLEARMASK) - bbStart;
    shifts = 0;

    while ((bbSize >> shifts) > 65534) shifts++;
  #end_block

  for (j = 0; j < bbSize; j++) {
    Int32 a2update = zptr[bbStart + j];
    UInt16 qVal = (UInt16)(j >> shifts);
  }
}\end{verbatim}
quadrant[a2update] = qVal;
if (a2update < NUM_OVERSHOOT_BYTES)
    quadrant[a2update + last + 1] = qVal;
}

if (! ( ((bbSize-1) >> shifts) <= 65535 ))
    panic ( "sortIt" );

A.4 Crafty

Crafty is a chess game program included in SPEC2000. The program intensively searches and evaluates the scores of potential moves using the Evaluate function. However, this process can potentially incur redundant computation on rare events, for example, trapping bishops. Therefore, we attach data triggers only to the variables WhiteBishops and BlackPawns, which can affect the evaluation regarding if a bishop is trapped.

In this section, we only demonstrate the modification for reducing redundant computation for trapping white bishops. Users can also apply similar idea to BlackBishops and WhitePawns for black bishops. The DTT model computes the update_WhiteBishops support thread function when any data trigger changes. However, the original code in the Evaluate function is not idempotent since it accumulates the result into the score variable. We make the update_WhiteBishops function idempotent by using a global variable trappingWhiteBishop to store the value to subtract for this part of the computation. We also need to change the code in the skippable region to make the code in the skippable region idempotent.

The following code sections present our implementation, including the data trigger declaration, the support thread function, and the modified skippable region.
Data trigger declaration

BITBOARD WhiteBishops; #trigger update_WhiteBishops();
BITBOARD BlackPawns; #trigger update_WhiteBishops();

The support thread function

#DTT update_WhiteBishops
void update_WhiteBishops(void *e)
{
    register BITBOARD temp;
    register int score=0;
    temp=And(WhiteBishops,mask_A7H7);
    while(temp) {
        square=FirstOne(temp);
        if (square == A7 && And(mask_B6B7,BlackPawns)) {
            if (And(set_mask[B6],BlackPawns) || Swap(B7,B6,0)>=0)
                score-=BISHOP_TRAPPED;
        } else if (square == H7 && And(mask_G6G7,BlackPawns)) {
            if (And(set_mask[G6],BlackPawns) || Swap(G7,G6,0)>=0)
                score-=BISHOP_TRAPPED;
        }
        Clear(square,temp);
    }
    trappingWhiteBishop = score;
}

The skippable region

#block update_WhiteBishops
trappingWhiteBishop = 0;
temp=And(WhiteBishops,mask_A7H7);
while(temp) {
    square=FirstOne(temp);
    if (square == A7 &&
    And(mask_B6B7,BlackPawns)) {
        if (And(set_mask[B6],BlackPawns)||
            Swap(B7,B6,0)>=0)
            trappingWhiteBishop-=BISHOP_TRAPPED;
    } else if (square == H7 &&
And(mask_G6G7,BlackPawns) {
    if (And(set_mask[G6],BlackPawns)
        || Swap(G7,G6,0)>=0)
        trappingWhiteBishop-=BISHOP_TRAPPED;
}
Clear(square,temp);
}

A.5 Eon

Eon is a ray tracer written in C++. The ray tracer sends several rays into a 3D object and simulates that path of these rays after encountering the object. The mrSurfaceList::viewingHit method in this application generates lots of redundant computation to reinitialize two variables, a_MR and a_VHR, even though their values remain the same after the computation in the following for-loop.

    for (int i = 0; i < length(); i++) {
        mrSurface *sPtr = surfaces[i];
        ggMaterialRecord a_MR;
        mrViewingHitRecord a_VHR;
        a_MR.UV = uvTemp;
        if ( ((!sPtr->boundingBox(time, time, objectBox)) ||
            ggOverlapBox3(rayBox,objectBox)) &&
            sPtr->viewingHit( r, time, tmin, a_tmax, a_VHR, a_MR))
            if (a_VHR.t < VHR.t) {
                hit_one = ggTrue;
                VHR = a_VHR;
                a_tmax = VHR.t;
                MR = a_MR;
            }
    }

In our implementation, we would like to avoid the re-initialization of a_MR and a_VHR. Therefore, we move the declaration of these two variables to the beginning of the mrSurfaceList::viewingHit method and make these variables data triggers. If the viewingHit method touches the content of a_MR or a_VHR, the DTT model executes the
corresponding support thread function to reinitialize the values. With this modification, the code only needs to initialize these variables when touched.

We summarize the modification below:

Data trigger declaration

```
ggMaterialRecord a_MR;
#trigger ggMaterialRecord::ggMaterialRecordThread();

mrViewingHitRecord a_VHR;
#trigger mrViewingHitRecord::mrViewingHitRecordThread();
```

The support thread function

```
#DTT ggMaterialRecordThread
ggMaterialRecordThread()
{ BRDFPointer = 0; kBRDF.Set(1.0);
    hasRay1 = hasRay2 = ggFalse; }

#DTT mrViewingHitRecordThread
mrViewingHitRecordThread() : p(ggFalse), UVW(ggFalse), UV(ggFalse)
{ hasUVW = hasUV = hasEmit = hasAdd = ggFalse; coverage = 1.0; }
```

The skippable region

```
ggMaterialRecord() {
    #block ggMaterialRecordThread
    BRDFPointer = 0; kBRDF.Set(1.0);
    hasRay1 = hasRay2 = ggFalse;
    #end_block
}

mrViewingHitRecord() : p(ggFalse), UVW(ggFalse), UV(ggFalse) {
    #block mrViewingHitRecordThread
    hasUVW = hasUV = hasEmit = hasAdd = ggFalse; coverage = 1.0;
    #end_block
}
```

A.6 equake

Equake simulates the propagation of earthquake waves in large valleys or basins. The time integration loop is the most critical part of this application. The profiling
result indicates that the application creates a significant amount of redundant loads when calling phi0, phi1, and phi2 functions in the third nested for-loop shown below:

```c
for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
        disp[disptplus][i][j] +=
            2.0 * M[i][j] * disp[dispt][i][j] -
            (M[i][j] - Exc.dt / 2.0 * C[i][j]) * 
            disp[disptminus][i][j] -
            Exc.dt * Exc.dt * 
            (M23[i][j] * phi2(time) / 2.0 +
             C23[i][j] * phi1(time) / 2.0 +
             V23[i][j] * phi0(time) / 2.0);
```

These three functions all use the variable `time` as inputs and only depend on the value of `time` and another variable, `Exc.t0`, to calculate the output. These three functions incur redundant computation, because the value of `time` only changes in the beginning of every loop iteration, and the value of `Exc.t0` never changes after it’s initialized.

Therefore, we attach data triggers to `time`. Since the current DTT model only allows one variable to associate with a support thread function, we create two other variables, `time1` and `time2`, to trigger support thread functions for phi1 and phi2. We also insert statements to set the values of phi1 and phi2 after the statement that can assign a new value to phi. The corresponding support thread function of each variable will perform the required computation and keep the result in a global variable. When the application visits these phi functions, the DTT model uses the stored result instead of recomputing values to reduce redundant computation.

The following code shows our modifications.

**Data trigger declaration**

```c
double time; #trigger update_phi0();
double time1; #trigger update_phi1();
```
double time2; #trigger update_phi2();

The support thread function

#block update_phi0
void phi0(double *t) {
    double value;
    if (t <= Exc.t0) {
        value = 0.5 / PI * (2.0 * PI * t / Exc.t0 - sin(2.0 * PI * t / Exc.t0));
        phi0_value = value;
    }
    else
        phi0_value = 1.0;
}
#end_block

#block update_phi1
void phi1_thread(double *t) {
    if (*t <= Exc.t0) {
        value = (1.0 - cos(2.0 * PI * *t / Exc.t0)) / Exc.t0;
        phi1_value = value;
    }
    else
        phi1_value = 0.0;
}
#end_block

#block update_phi2
void phi2_thread(double *t) {
    double value;
    if (*t <= Exc.t0) {
        value = 2.0 * PI / Exc.t0 / Exc.t0 * sin(2.0 * PI * *t / Exc.t0);
        phi2_value = value;
    }
    else
        phi2_value = 0.0;
}
#end_block
double phi0(t)
double t;
{
    double value;
    #block update_phi0
        if (t <= Exc.t0) {
            value = 0.5 / PI * (2.0 * PI * t / Exc.t0 -
                sin(2.0 * PI * t / Exc.t0));
            return value;
        }
    else
        return 1.0;
    #end_block
    return phi0_value;
}

double phi1(t)
double t;
{
    double value;
    #block update_phi1
        if (t <= Exc.t0) {
            value = (1.0 - cos(2.0 * PI * t / Exc.t0)) / Exc.t0;
            return value;
        }
    else
        return 0.0;
    #end_block
    return phi1_value;
}

double phi2(t)
double t;
{
    double value;
    #block update_phi2
        if (t <= Exc.t0) {
            value = 2.0 * PI / Exc.t0 / Exc.t0 * 
                sin(2.0 * PI * t / Exc.t0);
            return value;
        }
    }
else
    return 0.0;
#end_block
    return phi2_value;
}

A.7 gcc

The gcc benchmark included in the SPEC2000 is a C compiler based on gcc-2.7.2.2. For this benchmark, we also have different implementations for Chapter 4 and Chapter 5.

Our implementation in Chapter 4 found that in the program phase we simulated, the value returned from the \texttt{max\_reg\_num} never changes since the value of \texttt{reg\_rtx\_no} remains the same all the time. Therefore, we declare the \texttt{reg\_rtx\_no} as the trigger. If the value of \texttt{reg\_rtx\_no} changes, the support thread function \texttt{change\_max\_reg\_num} updates the recorded return value in the TST and injects this value to the register instead of accessing the memory.

The following code presents our modification for Chapter 4.

\textbf{Data trigger declaration}

\begin{verbatim}
int reg_rtx_no; #trigger change_max_reg_num();
\end{verbatim}

\textbf{The support thread function}

\begin{verbatim}
#DTT maxRegNum
int change_max_reg_num () {
    return reg_rtx_no;
}
\end{verbatim}

\textbf{The skippable region}

\begin{verbatim}
int max_reg_num () {
    #block maxRegNum
\end{verbatim}
return reg_rtx_no;
#endif
}

For the implementation used in Chapter 5, the profiler suggests that the propagate_block function is the most frequently executed function and contains potential for redundant computation – 97% of accesses to basic_block_live_at_end are redundant in this function. Therefore, we modify the declaration of basic_block_live_at_end by changing it from a pointer to an array and allow the DTT model to trigger support thread functions when an element in the array changes. The DTT model can skip the computation in the life_analysis function that calls the propagate_block function if the data trigger remains the same.

We demonstrate our implementation in Chapter 5 in the following code.

Data trigger declaration

regset basic_block_live_at_end[max_n_basic_blocks];
#trigger propagate_block_thread

The support thread function

#elif propagateBlock
void *propagate_block_thread (regset *basic_block_live_at_end_i)
{
    int i = basic_block_live_at_end_i - &basic_block_live_at_end[0]
    propagate_block (basic_block_live_at_end[i],
                     basic_block_head[i], basic_block_end[i], 1,
                     (regset) 0, i);

    #ifdef USE_C_ALLOCA
    alloca (0);
    #endif
    return NULL;
}

The skippable region

if (n_basic_blocks > 0)
for (i = FIRST_PSEUDO_REGISTER; i < max_regno; i++)
    if (basic_block_live_at_start[0][i / REGSET_ELT_BITS]
        & ((REGSET_ELT_TYPE) 1 << (i % REGSET_ELT_BITS)))
        reg_basic_block[i] = REG_BLOCK_GLOBAL;

max_scratch = 0;
#endif

A.8 gzip

gzip is a popular compression program. In this application, the percentage of redundant loads is high. The source of redundant loads is loads from relatively stable tables. However, the inputs generated after each iteration of the core algorithm are highly dynamic, and as a result, the percentage of silent stores is lower than 30%. In this benchmark, we can only apply the DTT model to accelerate a small piece of code in the longest_match function.

In Chapter 4, we focus on skipping the computation in the beginning parts of the function. This part of computation only depends on the value of two variables, hash_head and strstart. We attach the data trigger declarations to these two variables and trigger the support thread function, longest_match_thread if necessary. The DTT model can skip the beginning parts that calculates the scan, strend, and limit. The following C code shows our implementation.
Data trigger declaration

    int hash_head; #trigger longest_match_thread();
    int strstart; #trigger longest_match_thread();

The support thread function

    #DTT longestMatch
    void longest_match_thread()
    {
        scan = window + strstart;
        strend = window + strstart + MAX_MATCH;
        limit = strstart > (IPos)MAX_DIST ?
                strstart - (IPos)MAX_DIST : NIL;
    }

The skippable region

    int longest_match(cur_match)
    IPos cur_match;  {
    #block longestMatch
        scan = window + strstart; /* current string */
        strend = window + strstart + MAX_MATCH;
        limit =
            strstart > (IPos)MAX_DIST ? strstart - (IPos)MAX_DIST : NIL;
    #end_block

    For Chapter 5, we instead focus on another part in the longest_match function because the implementation in Chapter 4 provides no benefit in the software DTT. For this piece of code, we use the variable max_chain_length as the data trigger. As soon as the max_chain_length changes, the DTT model triggers the support thread function – update_variable_thread function. This allows us to skip some computation in the longest_match function.

    We use the following C code in Chapter 5.

Data trigger declaration

    unsigned near max_chain_length; #trigger update_variable_thread();
The support thread function

```c
#DTT updateVariable
void update_variable_thread()
{
    chain_length = max_chain_length; /* max hash chain length */
    best_len = prev_length; /* best match length so far */
    scan_end1 = scan[best_len-1];
    scan_end = scan[best_len];

    return;
}
```

The skippable region

```c
#undef UNALIGNED_OK
#ifdef UNALIGNED_OK
    register uch *strend = window + strstart + MAX_MATCH - 1;
    register ush scan_start = *(ush*)scan;
    register ush scan_end = *(ush*)(scan+best_len-1);
#else
#endif
```

A.9 mcf

For our implementation in mcf, please refer to Section 3.2.4.

A.10 mesa

Mesa is an OpenGL library that creates 3D objects from 2D scalar fields. For this benchmark, though both our implementation in Chapter 4 and Chapter 5 focus on the same function, our profiling in Chapter 5 covers a longer execution phase and allows
us to discover different opportunities for triggering computation using the DTT model.

In chapter 4, we found that one of the most frequently executed functions
\(-\text{sample\_1d\_linear}\) incurs high redundancy in both loads and stores because the inputs
\(i_0, i_1, w_0,\) and \(w_1\) do not dramatically change, and generate similar results between
invocations. Therefore, we declare four global variables to store the value of the four
inputs of the
\(\text{sample\_1d\_linear}\) and attach the data triggers to these variables.

We also add statements to assign values from local variables to global variables.
The DTT model triggers the \(\text{sample\_1d\_linear\_thread}\) whenever the contents of these
global variables change. Finally, we define part of the computation that the support
thread function can replace as the skippable region.

The code shown below presents our implementation in Chapter 4.

**Data trigger declaration**

| GLint i0_thread; #trigger sample_1d_linear_thread(); |
| GLint i1_thread; #trigger sample_1d_linear_thread(); |
| GLint w0_thread; #trigger sample_1d_linear_thread(); |
| GLint w1_thread; #trigger sample_1d_linear_thread(); |

**The support thread function**

```c

#DTT sample1DLinear
void sample_1d_linear_thread(GLint *x)
{
    GLubyte red0, green0, blue0, alpha0;
    GLubyte red1, green1, blue1, alpha1;
    if (i0border_thread) {
        red0 = tObj->BorderColor[0];
        green0 = tObj->BorderColor[1];
        blue0 = tObj->BorderColor[2];
        alpha0 = tObj->BorderColor[3];
    } else {
```
get_1d_texel(tObj, img, i0_thread, &red0, &green0, &blue0, &alpha0);
}
if (i1border_thread) {
    red1 = tObj->BorderColor[0];
    green1 = tObj->BorderColor[1];
    blue1 = tObj->BorderColor[2];
    alpha1 = tObj->BorderColor[3];
}
else {
    get_1d_texel(tObj, img, i1_thread, &red1, &green1, &blue1, &alpha1);
}

*red = (w0_thread*red0 + w1_thread*red1) >> 8;
*green = (w0_thread*green0 + w1_thread*green1) >> 8;
*blue = (w0_thread*blue0 + w1_thread*blue1) >> 8;
*alpha = (w0_thread*alpha0 + w1_thread*alpha1) >> 8;

The skippable region

static void sample_1d_linear( const struct gl_texture_object *tObj,
                               const struct gl_texture_image *img,
                               GLfloat s,
                               GLubyte *red, GLubyte *green,
                               GLubyte *blue, GLubyte *alpha )
{
    GLint width = img->Width2;
    GLint i0, i1;
    GLfloat u;
    GLint i0border, i1border;

    u = s * width;
    if (tObj->WrapS == GL_REPEAT) {
        i0 = ((GLint) floor(u - 0.5F)) % width;
        i1 = (i0 + 1) & (width-1);
        i0border = i1border = 0;
    }
    else {
        i0 = (GLint) floor(u - 0.5F);
        i1 = i0 + 1;
    }
i0border = (i0<0) | (i0>=width);
i1border = (i1<0) | (i1>=width);
}

if (img->Border) {
    i0 += img->Border;
i1 += img->Border;
i0border = i1border = 0;
} else {
i0 &= (width-1);
}

#define sample1DLinear
{
    GLfloat a = frac(u - 0.5F);
    GLint w0 = (GLint) ((1.0F-a) * 256.0F);
    GLint w1 = (GLint) ( a * 256.0F);

    GLubyte red0, green0, blue0, alpha0;
    GLubyte red1, green1, blue1, alpha1;

    if (i0border) {
        red0 = tObj->BorderColor[0];
        green0 = tObj->BorderColor[1];
        blue0 = tObj->BorderColor[2];
        alpha0 = tObj->BorderColor[3];
    } else {
        get_1d_texel( tObj, img, i0, &red0, &green0, &blue0, &alpha0 );
    }

    if (i1border) {
        red1 = tObj->BorderColor[0];
        green1 = tObj->BorderColor[1];
        blue1 = tObj->BorderColor[2];
        alpha1 = tObj->BorderColor[3];
    } else {
        get_1d_texel( tObj, img, i1, &red1, &green1, &blue1, &alpha1 );
    }

    *red = (w0*red0 + w1*red1) >> 8;
    *green = (w0*green0 + w1*green1) >> 8;
In Chapter 5, the profiling tool covers the execution of the whole program. We found that the redundant computation in the sample_1d_linear function originates from the several arrays – red, green, blue, and alpha in the general_textured_triangle function. We declare these arrays as data triggers and implement different support thread functions to perform incremental algorithms that updates the result for the sample_1d_linear function.

In the following code, we demonstrate the data trigger declaration, the support thread function, the get_1d_texel_red_thread that we created for the support thread function, and the skippable region code for the case when an element in the red array changes. The programmer can apply similar techniques to other arrays in addition to the data trigger that we use in this work.

**Data trigger declaration**

```c
GLubyte red[MAX_WIDTH]; #trigger sample_1d_linear_red_thread();
```

**The support thread function**

```c
#DTT sample1DLinear
void sample_1d_linear_red_thread(GLfloat *x)
{
    GLfloat s = *x;
    struct gl_texture_object *tObj = img_thread;
    struct gl_texture_image *img = tObj->Image[0];
    GLubyte red0;
    GLubyte red1;
    GLint width = img->Width2;
    GLint i0, i1, i0border, i1border;
    GLfloat u = s * width;
    GLfloat a = frac(u - 0.5F);
```
GLint w0 = (GLint) ((1.0F-a) * 256.0F);
GLint w1 = (GLint) (a * 256.0F);

if (tObj->WrapS==GL_REPEAT) {
    i0 = ((GLint) floor(u - 0.5F)) % width;
    i1 = (i0 + 1) & (width-1);
    i0border = i1border = 0;
} else {
    i0 = (GLint) floor(u - 0.5F);
    i1 = i0 + 1;
    i0border = (i0<0) | (i0>=width);
    i1border = (i1<0) | (i1>=width);
}

switch (img->Format) {
case GL_LUMINANCE:
case GL_INTENSITY:
    if (i0border) {
        red0 = tObj->BorderColor[0];
    }
    else {
        get_1d_texel_red_thread(tObj, img, i0, &red0);
    }
    if (i1border) {
        red1 = tObj->BorderColor[0];
    }
    else {
        get_1d_texel_red_thread(tObj, img, i1, &red1);
    }
    lastRed = (w0*red0 + w1*red1) >> 8;
    return;
}

The get_1d_texel_red_thread function

void get_1d_texel_red_thread( const struct gl_texture_object *tObj,
                               const struct gl_texture_image *img,
                               GLint i,
                               GLubyte *red)
{
    GLubyte *texel;
#ifdef DEBUG
GLint width = img->Width;
if (i<0 || i>=width) abort();
#endif

switch (img->Format) {
    case GL_COLOR_INDEX:
    {
        GLubyte index = img->Data[i];
        palette_sample(tObj, index, red, green, blue, alpha);
        return;
    }
    return;
    case GL_ALPHA:
        *alpha = img->Data[i];
        return; /*
    case GL_LUMINANCE:
    case GL_INTENSITY:
        *red = img->Data[i];
        return;
    case GL_LUMINANCE_ALPHA:
        texel = img->Data + i * 2;
        *red = texel[0];
        return;
    case GL_RGB:
        texel = img->Data + i * 3;
        *red = texel[0];
        return;
    case GL_RGBA:
        texel = img->Data + i * 4;
        *red = texel[0];
        return;
    default:
        gl_problem(NULL, "Bad format in get_1d_texel");
        return;
}

The skippable region

static void sample_1d_linear( const struct gl_texture_object *tObj,
    const struct gl_texture_image *img,
    GLfloat s,
    GLubyte *red, GLubyte *green,
GLubyte *blue, GLubyte *alpha )
{

GLint width = img->Width2;
GLint i0, i1;
GLfloat u;
GLint i0border, i1border;

u = s * width;
if (tObj->WrapS==GL_REPEAT) {
    i0 = ((GLint) floor(u - 0.5F)) % width;
    i1 = (i0 + 1) & (width-1);
    i0border = i1border = 0;
}
else {
    i0 = (GLint) floor(u - 0.5F);
    i1 = i0 + 1;
    i0border = (i0<0) | (i0>=width);
    i1border = (i1<0) | (i1>=width);
}

if (img->Border) {
    i0 += img->Border;
    i1 += img->Border;
    i0border = i1border = 0;
}
else {
    i0 &= (width-1);
}

#define sample1DLinear
{

    GLfloat a = frac(u - 0.5F);

    GLint w0 = (GLint) ((1.0F-a) * 256.0F);
    GLint w1 = (GLint) (a * 256.0F);

    GLubyte red0, green0, blue0, alpha0;
    GLubyte red1, green1, blue1, alpha1;

    if (i0border) {
        red0 = tObj->BorderColor[0];
        green0 = tObj->BorderColor[1];
        blue0 = tObj->BorderColor[2];
        alpha0 = tObj->BorderColor[3];
    }
    else {

get_1d_texel
( tObj, img, i0, &red0, &green0, &blue0, &alpha0 );
}
if (i1border) {
    red1 = tObj->BorderColor[0];
    green1 = tObj->BorderColor[1];
    blue1 = tObj->BorderColor[2];
    alpha1 = tObj->BorderColor[3];
}
else {
    get_1d_texel
( tObj, img, i1, &red1, &green1, &blue1, &alpha1 );
}

*red = (w0*red0 + w1*red1) >> 8;
*green = (w0*green0 + w1*green1) >> 8;
*blue = (w0*blue0 + w1*blue1) >> 8;
*alpha = (w0*alpha0 + w1*alpha1) >> 8;
}
return;
#endif

A.11 parser

Parser is an English syntactic parser. In parser, the loads that access the hash table in the count function are 99% redundant, on average. However, these redundant loads do not necessarily lead to redundant computation. The computation in the count function depends on two inputs and the same pair of inputs never appear twice. Though almost every access to the hash table is redundant, the following computation is not redundant.

In parser, we can only take advantage from triggering the hash table lookup earlier once we know the inputs of the function. However, because the DTT model only
allows one argument, we create global variables to store all the inputs of the count function and make these variables data triggers. In the beginning of the count function, we assign the arguments into these variables that we added for the DTT model and trigger the support thread function – hash_thread. Finally, we also need to define the original code in the hash function that looks up the randtable as the skippable region. Again, because this application does not contain the type of redundant computation the DTT model can exploit, our modification always increases the dynamic instructions.

We list the code here.

Data trigger declaration

```c
int lw_thread; #trigger hash_thread();
int rw_thread; #trigger hash_thread();
Connector *le_thread; #trigger hash_thread();
Connector *re_thread; #trigger hash_thread();
int cost_thread; #trigger hash_thread();
```

The support thread function

```c
void hash_thread(void *x){
    i = i + (i<<1) + randtable[(lw_thread + i) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(rw_thread + i) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(((long) le_thread + i) % (table_size_thread+1)) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(((long) re_thread + i) % (table_size_thread+1)) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(cost_thread+i) & (RTSIZE - 1)];
    i_DTT = i;
}
```

The skippable region

```c
int hash(int lw, int rw, Connector *le, Connector *re, int cost) {
    int i;
```
#block hashLookup
    i = 0;
    i = i + (i<<1) + randtable[(lw + i) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(rw + i) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(((long) le + i) % (table_size+1)) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(((long) re + i) % (table_size+1)) & (RTSIZE - 1)];
    i = i + (i<<1) + randtable[(cost+i) & (RTSIZE - 1)];
    i_DTT = i;
#end_block
    return i_DTT & (table_size-1);
}

A.12 perlbmk

The benchmark version of Perl in SPEC2000 implements the core Perl interpreter and several third party modules. Though this benchmark reveals 86% redundant loads and 70% redundant stores in Chapter 2, most of this redundant behavior comes from register spilling rather than real redundant behavior. In Chapter 4, we can only apply the DTT model to precompute the interpretation when the operation of a perl statement is determined. We implement the runops_standard_thread function to pre-execute the function needed to run on each operation, and store the result to the global variable next_op. In the original runops_standard function, we can skip the while-loop and assign the stored dtt_op variable to PL_op if the DTT model has already done so.

Here are our modifications in Chapter 4.

Data trigger declaration

    PERLVAR(Top,OP *) #trigger runops_standard_thread()
The support thread function

```c
#DTT runOps
void runops_standard_thread(OP *x) {
    dTHR;
    OP *dtt_op = x;
    while ( dtt_op = (dtt_op->op_ppaddr)(ARGS) ) {
        TAINT_NOT;
        next_op = dtt_op;
    }
}
```

The skippable region

```c
int
runops_standard(void)
{
    dTHR;
    #block runOp
    while ( PL_op = (CALLOP->op_ppaddr)(ARGS) ) {
        TAINT_NOT;
        return 0;
    }
    #end_block
    PL_op = dtt_op;
    return 0;
}
```

In Chapter 5, we found that the multithreading overhead makes the implementation in Chapter 4 inefficient. We instead change the `gv_stashpvn` function only needs to update its result when the name passing into the function changes. Therefore, we attach the data trigger to a global pointer to the name string. When the application assigns a new name string to the pointer, the DTT model executes the `gv_stashpvn_thread` support thread function. Once the support thread function finishes, the application does not need to recompute the `gv_stashpvn` again.

The following code demonstrates our modification in Chapter 5.
Data trigger declaration

char *name_thread; #trigger gv_stashpvn_thread();

The support thread function

#DTT gvStashpvn
void* gv_stashpvn_thread(void *x) {
    char smallbuf[256];
    char *tmpbuf;
    HV *stash;
    GV *tmpgv;
    lastnamelen = strlen(name_thread);
    if (namelen_thread + 3 < sizeof smallbuf)
        tmpbuf = smallbuf;
    else
        New(606, tmpbuf, namelen_thread + 3, char);
    Copy(name_thread,tmpbuf,namelen_thread,char);
    tmpbuf[namelen_thread++] = ':';
    tmpbuf[namelen_thread++] = ':';
    tmpbuf[namelen_thread] = '\0';
    tmpgv = gv_fetchpv(tmpbuf, create_thread, SVt_PVHV);
    if (tmpbuf != smallbuf)
        Safefree(tmpbuf);
    if (!tmpgv)
        return 0;
    if (!GvHV(tmpgv))
        GvHV(tmpgv) = newHV();
    stash = GvHV(tmpgv);
    if (!HvNAME(stash))
        HvNAME(stash) = savepv(name_thread);
    lastStash = stash;
}
\subsubsection{The skippable region}

gv_stashpvn(char *name, U32 namelen, I32 create) {
    char smallbuf[256];
    char *tmpbuf;
    HV *stash;
    GV *tmpgv;
    namelen_thread = namelen;
    name_thread = name;
A.13 twolf

Twolf in SPEC2000 uses the "TimberWolfSC" package to determine the placement of transistors. Several of the load instructions in one of the most time-consuming functions – new_dbox_a reveals high percentage of redundant loads in our profiling result. These redundant loads come from loading pointer addresses, but do not lead to the type of redundant computation that the DTT model can eliminate. Therefore, in our implementation, we simply use the DTT model to trigger the computation of the new_dbox_a function as soon as the value in the new_total field of the dimbox structure changes. The changing of this field indicates that the data structure is modified and will be performing the new_dbox_a function to update the cost field. In the support thread function, we update cost field of the dimbox structure. Once the cost is calcul-
lated and remains valid before the main thread visits the code, the DTT model can skip
the update of the cost field in each net and add all cost fields.

We list the modified code.

**Data trigger declaration**

```c
typedef struct dimbox {
    int new_total ; #trigger update_new_dbox_a();
}
*DBOXPTR , DBOX ;
```

**The support thread function**

```c
#DTT update_new_dbox_a
void *update_new_dbox_a(DBOXPTR x)
{
    DBOXPTR dimptr ;
    NBOXPTR netptr ;
    int old_mean , new_mean , oldx , newx ;
    int min , max , row , net ;
    dimptr = x;
    if( dimptr->dflag == 0 ) {
        return;
    }
    new_mean = dimptr->new_total / dimptr->numpins ;
    old_mean = dimptr->old_total / dimptr->numpins ;
    dimptr->cost=0;
    for( netptr = dimptr->netptr ; netptr ;
        netptr = netptr->nterm )
    {
        oldx = netptr->xpos ;
        if( netptr->flag == 1 ) {
            netptr->flag = 0 ;
            newx = netptr->newx ;
        } else {
            newx = oldx ;
        }
        dimptr->cost +=
        ABS( newx - new_mean ) - ABS( oldx - old_mean ) ;
        dimptr->dflag = 1;
        return ;
    }
```
The skippable region

```c
new_dbox_a( antrmptr , costptr )
TEBOXPTR antrmptr ;
int *costptr ;
{
    for( termptr = antrmptr ; termptr ;
        termptr = termptr->nextterm )
    {
        net = termptr->net ;
        dimptr = netarray[ net ] ;
        if( dimptr->dflag == 0 ) {
            continue ;
        }
        #block update_new_dbox_a
        if((dimptr->dflag == 2)
        {
            new_mean = dimptr->new_total / dimptr->numpins ;
            old_mean = dimptr->old_total / dimptr->numpins ;
            dimptr->cost = 0;
            for( netptr = dimptr->netptr ; netptr ; netptr = netptr->nterm )
            {
                oldx = netptr->xpos ;
                if( netptr->flag == 1 ) {
                    newx = netptr->newx ;
                    netptr->flag = 0 ;
                } else {
                    newx = oldx ;
                }
                dimptr->cost +=
                ABS( newx - new_mean ) - ABS( oldx - old_mean );
            }
        }
        #end_block
        *costptr += dimptr->cost;
        dimptr->dflag = 0 ;
    }
}
```

A.14 VORTEX

VORTEX is database transaction benchmark. In this benchmark, we found many redundant loads in the most frequently executed Chunk_ChkGetChunk function. We
traced the code and found many of the calls to the Chunk_ChkGetChunk function were originated from PersonObjs_FindIn. Therefore, we focused on triggering the computation of Chunk_ChkGetChunk as soon as we detect any change to the inputs of PersonObjs_FindIn function by attaching data triggers to the inputs of the PersonObjs_FindIn function.

The DTT model triggers the computation that performs the computation of PersonObjs_FindIn using the support thread function update_PersonObjs_FindId, which generates calls to the Chunk_ChkGetChunk function and stores the result in the PersonObjs_FindIn_return variable. In addition, we also need to modify the PersonObjs_FindIn function to accommodate the addition of the PersonObjs_FindIn_return variable as shown below:

We shows our modification as below.

Data trigger declaration

```c
tokentype EmpTkn010; #trigger update_PersonObjs_FindId();
addrtype PersonId; #trigger update_PersonObjs_FindId();
tokentype PersonTkn;#trigger update_PersonObjs_FindId();
```

The support thread function

```c
#DTT update_PersonObjs_FindId
void update_PersonObjs_FindId(void *x)
{
    tokentype *OwnerTkn = &EmpTkn010;
    addrtype KeyValue = &PersonId;
    tokentype *MemberTkn = &PersonTkn;
    Owner_KEYSETFindIn (PersonObjs_Set, OwnerTkn, 
                       KeyValue, McStat, MemberTkn);
    PersonObjs_FindIn_return = STAT;
}
```
The skippable region

```c
boolean PersonObjs_FindIn(tokentype *OwnerTkn, addrtype KeyValue, ft F,lt Z,zz *Status, tokentype *MemberTkn)
{
    #block update_PersonObjs_FindId
    Owner_KeySetFindIn (PersonObjs_Set, OwnerTkn, KeyValue, McStat, MemberTkn);
    TRACK(TrackBak,"FindInPersonObjs
");
    PersonObjs_FindIn_return = STAT;
    #end_block
    return (PersonObjs_FindIn_return);
}
```

A.15  vpr

For vpr, a circuit placement and routing program, our modifications for Chapter 4 and Chapter 5 are different.

For Chapter 4, we worked on the add_to_heap function to trigger the my_realloc function as soon as we know the heap is full. We attach the data trigger to the heap_tail variable and judge if we need to invoke the my_realloc function. We also define the code region that invokes the my_realloc function as the skippable region.

**Data trigger declaration**

```c
static int heap_tail;   #trigger my_realloc_DTT();
```

**The support thread function**

```c
#DTT myRealloc
void *my_realloc_DTT(void *x)
{
    dtt_event *e = (dtt_event *)x;
    if (heap_tail > heap_size )
    {
        /* Heap is full */
```
heap_size *= 2;
heap = my_realloc ((void *)(heap + 1), heap_size *
    sizeof (struct s_heap *));
heap--; /* heap goes from [1..heap_size] */
}
}

The skippable region

static void add_to_heap (struct s_heap *hptr) {
/* Adds an item to the heap, expanding the heap if necessary.*/
int ito, ifrom;
#if 0
myRealloc
#endif
if (heap_tail > heap_size ) { /* Heap is full */
    heap_size *= 2;
    heap = my_realloc ((void *)(heap + 1), heap_size *
        sizeof (struct s_heap *));
    heap--; /* heap goes from [1..heap_size] */
}
#if 0
#endif
heap[heap_tail] = hptr;

For Chapter 5, the profiler reports check_node as the most opportune function to exploit redundant computation. We make several fields in the s_rr_node data structure as data triggers. This implementation triggers the DTT model to perform the computation in the check_node function only when one of the fields in the s_rr_node changes. Then, the DTT model can skip the call to check_node in the check_rr_graph if the support thread function already completed the computation.

We demonstrate our modification using the following C code.

Data trigger declaration

struct s_rr_node {
    short xlow; #trigger check_node_thread();
    short xhigh; #trigger check_node_thread();
    short ylow; #trigger check_node_thread();
    short yhigh; #trigger check_node_thread();
    short ptc_num; #trigger check_node_thread();
    short num_edges; #trigger check_node_thread();
The support thread function

```c
#DTT checkNode
void *check_node_thread (s_rr_node *x) {

    int inode = (int)(x-&rr_node[0]);
    enum e_route_type route_type = router_opts_thread->route_type;
    int xlow, ylow, xhigh, yhigh, ptc_num, capacity;
    t_rr_type rr_type;
    int nodes_per_chan, tracks_per_node, num_edges;

    rr_type = rr_node[inode].type;
    xlow = rr_node[inode].xlow;
    xhigh = rr_node[inode].xhigh;
    ylow = rr_node[inode].ylow;
    yhigh = rr_node[inode].yhigh;
    ptc_num = rr_node[inode].ptc_num;
    capacity = rr_node_cost_inf[inode].capacity;

    if (xlow > xhigh || ylow > yhigh) {
        printf ("Error in check_node: rr endpoints are (%d,%d) and (%d,%d).\n",
                xlow, ylow, xhigh, yhigh);
        exit (1);
    }

    if (xlow < 0 || xhigh > nx+1 || ylow < 0 || yhigh > ny+1) {
        printf ("Error in check_node: rr endpoints, (%d,%d) and (%d,%d), \n"       "are out of range.\n", xlow, ylow, xhigh, yhigh);
        exit (1);
    }

    if (ptc_num < 0) {
        printf ("Error in check_node. Inode %d (type %d) had a ptc_num\n"       "of %d.\n", inode, rr_type, ptc_num);
        exit (1);
    }
}
```
switch (rr_type) {

  case SOURCE: case SINK: case IPIN: case OPIN:
    if (xlow != xhigh || ylow != yhigh) {
      printf("Error in check_node: Node
"    "(%d,%d) and (%d,%d)\n", innode, rr_type, xlow, ylow, xhigh, yhigh);
      exit (1);
    }
    if (clb[xlow][ylow].type != CLB
&& clb[xlow][ylow].type != IO) {
      printf("Error in check_node: Node
"    " clb location (%d, %d).\n", innode, rr_type, clb[xlow][ylow].type, xlow, ylow);
    exit (1);
    }
  break;

  case CHANX:
    if (xlow < 1 || xhigh > nx || yhigh > ny || yhigh != ylow) {
      printf("Error in check_node: CHANX out of range.\n");
      printf("Endpoints: (%d,%d) and (%d,%d)\n", xlow, ylow, xhigh, yhigh);
      exit(1);
    }
    if (route_type == GLOBAL && xlow != xhigh) {
      printf("Error in check_node: node %d
spans multiple channel segments\n"    "which is not allowed with global routing.\n", innode);
      exit (1);
    }
  break;

  case CHANY:
    if (xhigh > nx || ylow < 1 || yhigh > ny || xlow != xhigh) {
      printf("Error in check_node: CHANY out of range.\n");
      printf("Endpoints: (%d,%d) and (%d,%d)\n", xlow, ylow, xhigh, yhigh);
      exit(1);
if (route_type == GLOBAL && ylow != yhigh) {
    printf ("Error in check_node: node %d spans multiple
    channel segments\n" "which is not
allowed with global routing.\n", inode);
    exit (1);
} else {
    break;
}

default:
    printf("Error in check_node: Unexpected segment
type: %d\n", rr_type);
    exit(1);
}

switch (rr_type) {

    case SOURCE:
        if (clb[xlow][ylow].type == CLB) {
            if (ptc_num >= num_class ||
                class_inf[ptc_num].type != DRIVER) {
                printf ("Error in check_node.
Inode %d (type %d) had a ptc_num\n"
                       "of %d.\n", inode, rr_type, ptc_num);
                exit (1);
            }
            if (class_inf[ptc_num].num_pins != capacity) {
                printf ("Error in check_node. Inode %d (type %d)
had a capacity\n"
                       "of %d.\n", inode, rr_type, capacity);
                exit (1);
            }
        }
        else { /* IO block */
            if (ptc_num >= io_rat) {
                printf ("Error in check_node. Inode %d (type %d)
had a ptc_num\n"
                       "of %d.\n", inode, rr_type, ptc_num);
                exit (1);
            }
            if (capacity != 1) {
                printf ("Error in check_node: Inode %d (type %d)
had a capacity\n"
                       "of %d.\n", inode, rr_type, capacity);
            }
        }
    }

else { /* IO block */
    if (ptc_num >= io_rat) {
        printf ("Error in check_node. Inode %d (type %d)
had a ptc_num\n"
               "of %d.\n", inode, rr_type, ptc_num);
        exit (1);
    }
    if (capacity != 1) {
        printf ("Error in check_node: Inode %d (type %d)
had a capacity\n"
               "of %d.\n", inode, rr_type, capacity);
    }
}
exit (1);
}
}
break;

case SINK:
  if (clb[xlow][ylow].type == CLB) {
    if (ptc_num >= num_class ||
        class_inf[ptc_num].type != RECEIVER) {
      printf("Error in check_node. Inode \%d (type \%d)
              had a ptc_num
"                     "of \%d.\n", inode, rr_type, ptc_num);
      exit (1);
    }
    if (class_inf[ptc_num].num_pins != capacity) {
      printf("Error in check_node. Inode \%d (type \%d)
              has a capacity
"                     "of \%d.\n", inode, rr_type, capacity);
      exit (1);
    }
  }
else { /* IO block */
    if (ptc_num >= io_rat) {
      printf("Error in check_node. Inode \%d (type \%d)
              had a ptc_num
"                     "of \%d.\n", inode, rr_type, ptc_num);
      exit (1);
    }
    if (capacity != 1) {
      printf("Error in check_node: Inode \%d (type \%d)
              has a capacity
"                     "of \%d.\n", inode, rr_type, capacity);
      exit (1);
    }
  }
break;

case OPIN:
  if (clb[xlow][ylow].type == CLB) {
    if (ptc_num >= pins_per_clb ||
        class_inf[clb_pin_class[ptc_num]].type
        != DRIVER) {
      printf("Error in check_node. Inode \%d (type \%d)
              had a ptc_num
"                     "of \%d.\n", inode, rr_type, ptc_num);
      exit (1);
    }
  }
exit (1);
}

else { /* IO block */
    if (ptc_num >= io_rat) {
        printf("Error in check_node. Inode %d (type %d)
           had a ptc_num\n" "of %d.\n", inode, rr_type, ptc_num);
        exit (1);
    }
}

if (capacity != 1) {
    printf("Error in check_node: Inode %d (type %d)
           has a capacity\n" "of %d.\n", inode, rr_type, capacity);
    exit (1);
}  
break;

case IPIN:
    if (clb[xlow][ylow].type == CLB) {
        if (ptc_num >= pins_per_clb ||
            class_inf[clb_pin_class[ptc_num]].type
            != RECEIVER) {
            printf("Error in check_node. Inode %d (type %d)
               had a ptc_num\n" "of %d.\n", inode, rr_type, ptc_num);
            exit (1);
        }
    }  
else { /* IO block */
        if (ptc_num >= io_rat) {
            printf("Error in check_node. Inode %d (type %d)
               had a ptc_num\n" "of %d.\n", inode, rr_type, ptc_num);
            exit (1);
        }
    }
    if (capacity != 1) {
        printf("Error in check_node: Inode %d (type %d)
               has a capacity\n" "of %d.\n", inode, rr_type, capacity);
        exit (1);
break;

case CHANX:
    if (route_type == DETAILED) {
        nodes_per_chan = chan_width_x[ylow];
        tracks_per_node = 1;
    }
    else {
        nodes_per_chan = 1;
        tracks_per_node = chan_width_x[ylow];
    }

    if (ptc_num >= nodes_per_chan) {
        printf("Error in check_node: Inode %d (type %d) has a ptc_num\n"
             "of %d.\n", inode, rr_type, ptc_num);
        exit (1);
    }

    if (capacity != tracks_per_node) {
        printf("Error in check_node: Inode %d (type %d) has a capacity\n"
             "of %d.\n", inode, rr_type, capacity);
        exit (1);
    }
    break;

case CHANY:
    if (route_type == DETAILED) {
        nodes_per_chan = chan_width_y[xlow];
        tracks_per_node = 1;
    }
    else {
        nodes_per_chan = 1;
        tracks_per_node = chan_width_y[xlow];
    }

    if (ptc_num >= nodes_per_chan) {
        printf("Error in check_node: Inode %d (type %d) has a ptc_num\n"
             "of %d.\n", inode, rr_type, ptc_num);
        exit (1);
    }
if (capacity != tracks_per_node) {
    printf("Error in check_node: Inode %d (type %d)
"%
           "has a capacity of %d.\n", inode, rr_type, capacity);
    exit (1);
} 
break;

default:
    printf("Error in check_node: Unexpected segment type:
%d\n", rr_type);
    exit(1);
}

num_edges = rr_node[inode].num_edges;

if (rr_type != SINK) {
    if (num_edges <= 0) {
        printf("Error in check_node: node %d has no edges.\n", inode);
        exit (1);
    }
}

else { /* SINK -- remove this check if feedthroughs allowed */
    if (num_edges != 0) {
        printf("Error in check_rr_graph: node %d is a sink, but has "
"%d edges.\n", inode, num_edges);
        exit (1);
    }
}

The skippable region

for (inode=0; inode<num_rr_nodes; inode++) {
    num_edges = rr_node[inode].num_edges;
#block checkNode
    check_node (inode, route_type);
#end_block
    rr_type = rr_node[inode].type;
    C = rr_node[inode].C;
A.16 blackscholes

Black-scholes is a financial application that evaluates stock options using the Black-Scholes Partial Differential Equation. The bs_thread function in this benchmark contains a for-loop that invokes BlkSchlsEqEuroNoDiv 100 times (defined by NUM_RUNS). Our profiling tool indicates that the inputs remain the same between iterations. Therefore, we declare all the six array inputs of the BlkSchlsEqEuroNoDiv function as data triggers. Each different array would trigger a different support thread function. For example, any change to the sptprice array triggers the bsInnerLoopDTTSptPrice function that calls the BlkSchlsEqEuroNoDiv function and updates the prices array that stores the computation result. After all support thread functions finish, the main thread does not need to call BlkSchlsEqEuroNoDiv function. Therefore, we skip the computation of the BlkSchlsEqEuroNoDiv function by making the statement in bs_thread function as the skippable region.

We list the code snippets here.

Data trigger declaration

```c
// Definition of arrays
int otype[10000000];  //trigger bsInnerLoopDTT0type();
ftype sptprice[10000000];  //trigger bsInnerLoopDTTSptPrice();
ftype strike[10000000];  //trigger bsInnerLoopDTTStrike();
ftype rate[10000000];  //trigger bsInnerLoopDTTRate();
ftype volatility[10000000];  //trigger bsInnerLoopDTTVolatility();
ftype otime[10000000];  //trigger bsInnerLoopDTT0time();
```

The support thread function

```c
#DTT bsThreadInnerLoop
void bsInnerLoopDTTSptPrice(ftype *dataPtr) {
```
// Get the index value
int i = dataPtr - &sptprice[0];
ftype price;

price = BlkSchlsEqEuroNoDiv( sptprice[i], strike[i],
                           rate[i], volatility[i], otime[i],
                           otype[i], 0);

prices[i] = price;
}

The skippable region

int bs_thread(void *tid_ptr) {
    int i, j;
    fptype price;
    fptype priceDelta;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);
    for (j=0; j<NUM_RUNS; j++) {
        #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.17 bodytrack

Bodytrack is a computer vision application that tracks a person within a series of bitmap files. For each camera image, this application uses Monte Carlo re-sampling to choose particles. Because of the randomness in the algorithm, the particles frequently change and the ratio of redundant computation is not high. Our DTT implementation for this benchmark targets using the support thread to calculate the Estimate
method as soon as a particle filter object gets updated. We added a field `updated` in the `ParticleFilter` class. This variable is incremented when the `Update` method finishes updating the `ParticleFilter`. If support thread function events complete, the loop in the main function does not need to call the `Estimate` function again, and our implementation marks that function call as the skippable region.

The following code sections contain the data trigger declaration, the modified `Update` function, the support thread function, and the skippable region.

**Data trigger declaration**

```cpp
template<class T>
class ParticleFilter{
public:
  //Types
  typedef float fpType;
  typedef std::vector<fpType> Vectorf;
  std::vector<Vectorf> mParticles, mNewParticles;
  //lists of particles
  Vectorf mWeights, mCdf;
  //particle weights, cumulative distribution
  int updated;  //trigger Estimate_thread();
protected:
  //variables
  T *mModel;
}
```

**The modified Update function**

```cpp
bool ParticleFilter<T>::Update(fpType timeval)
//weights have already been com
{
  if(!mInitialized)
    //check for proper initialization
    {
      std::cout << "Update Error : Particles not initialized" <<
      std::endl;
      return false;
    }
  if(!mModel->GetObservation(timeval))
    {
      std::cout << "Update Error : Model observation failed for time : "
      return false;
    }
```

```
for(int k = (int)mModel->StdDevs().size() - 1; k >= 0 ; k--)
    //loop over all annealing steps
    {  CalcCDF(mWeights, mCdf);
        //Monte Carlo re-sampling
        Resample(mCdf, mBins, mSamples, mNParticles);
        bool minValid = false;
        while(!minValid)
            {  GenerateNewParticles(k);
                CalcWeights(mNewParticles);
                //calculate particle weights an
                minValid = (int)mNewParticles.size() >= mMinParticles;
                //repeat if not enough valid pa
                if(!minValid)
                { std::cout << "Not enough valid particles - Resampling!!" << std::endl;
                }
                mParticles = mNewParticles;
            //save new particle set
            }
        updated++;
    }

The support thread function

#DTT Estimate
template<class T>
void ParticleFilter<T>::Estimate_thread(void *x)
{  
    estimate.assign(mParticles[0].size(), 0);
    for(uint i = 0; i < mParticles.size(); i++)
        for(uint j = 0; j < estimate.size(); j++)
            estimate[j] += mParticles[i][j] * mWeights[i];
}

The skippable region

for(int i = 0; i < frames; i++)
    //process each set
    {  cout << "Processing frame " << i << endl;
        if(!pf.Update((float)i))
// Run particle filt
{  cout << "Error loading observation data" << endl;
    workers.JoinAll();
    return 0;
}

#block Estimate
    pf.Estimate(estimate);
#end_block

// get average
WritePose(outputFileAvg, estimate);
if(OutputBMP)
    pf.Model().OutputBMP(estimate, i);

// save output bitma

A.18 canneal

Canneal implements the simulated annealing algorithm to minimize the cost of circuit routing. In the critical Run method, this application randomly picks one new node, calculates the cost of the new node and the old node, and then swaps the two nodes if that reduces the cost. However, if there is no swapping, evaluating the cost of the old node is redundant. Therefore, our implementation focuses on avoiding redundant computation when there is no swapping.

To achieve this goal, we implement a new method, the routing_cost_given_loc_p method, and modified the swap_cost method in the netlist_elem class to get the cost of an individual node. Since Run only picks one node in each iteration, we only update the cost when the method picks a new netlist_elem b and make this pointer a data trigger. When the DTT model computes the swap cost, the later code can reuse the computed cost fields of the node without recalculating them.

We list the data trigger declaration, the modified routing_cost_given_loc_p method, the modified swap_cost method, the modified Run method, the support thread
function, and the skippable region in the following C++ code.

**Data trigger declaration**

```cpp
netlist_elem* b; #trigger routing_cost_given_loc_thread();
```

**The routing_cost_given_loc_p method**

```cpp
routing_cost_t netlist_elem::routing_cost_given_loc_p (location_t* loc)
{
    routing_cost_t total_cost=0;
    for (int i = 0; i< fanin.size(); ++i){
        location_t* fanin_loc = fanin[i]->present_loc.Get();
        total_cost += fabs(loc->x - fanin_loc->x);
        total_cost += fabs(loc->y - fanin_loc->y);
    }
    for (int i = 0; i< fanout.size(); ++i){
        location_t* fanout_loc = fanout[i]->present_loc.Get();
        total_cost += fabs(loc->x - fanout_loc->x);
        total_cost += fabs(loc->y - fanout_loc->y);
    }
    return total_cost;
}
```

**The modified swap_cost method**

```cpp
routing_cost_t netlist_elem::swap_cost(location_t* old_loc, location_t* new_loc)
{
    routing_cost_t no_swap = 0;
    routing_cost_t yes_swap = 0;
    for (int i = 0; i< fanin.size(); ++i){
        location_t* fanin_loc = fanin[i]->present_loc.Get();
        no_swap += fabs(old_loc->x - fanin_loc->x);
        no_swap += fabs(old_loc->y - fanin_loc->y);
        yes_swap += fabs(new_loc->x - fanin_loc->x);
        yes_swap += fabs(new_loc->y - fanin_loc->y);
    }
    return yes_swap;
}
```
for (int i = 0; i < fanout.size(); ++i) {
    location_t* fanout_loc = fanout[i]->present_loc.Get();
    no_swap += fabs(old_loc->x - fanout_loc->x);
    no_swap += fabs(old_loc->y - fanout_loc->y);
    yes_swap += fabs(new_loc->x - fanout_loc->x);
    yes_swap += fabs(new_loc->y - fanout_loc->y);
}

new_cost = yes_swap;
old_cost = no_swap;
return yes_swap - no_swap;
}

The modified code in the Run method

if (is_good_move == move_decision_accepted_bad){
    accepted_bad_moves++;
    _netlist->swap_locations(a,b);
    b->old_cost = b->new_cost;
} else if (is_good_move == move_decision_accepted_good){
    accepted_good_moves++;
    _netlist->swap_locations(a,b);
    b->old_cost = b->new_cost;
} else if (is_good_move == move_decision_rejected){
    //no need to do anything for a rejected move
}

The support thread function

void routing_cost_given_loc_thread(void *x) {
    location_t* loc = last_b->present_loc.Get();
    last_b->swap_cost(loc, last_a->present_loc.Get());
}

The skippable region

routing_cost_t annealer_thread::
calculate_delta_routing_cost(netlist_elem* a, netlist_elem* b) {
    location_t* a_loc = a->present_loc.Get();
}
location_t* b_loc = b->present_loc.Get();

routing_cost_t delta_cost = 0;
  a->new_cost = a->routing_cost_given_loc_p(b_loc);
delta_cost += a->new_cost;
#block deltaCost
  delta_cost -= a->routing_cost_given_loc_p(a_loc);
  delta_cost += b->swap_cost(b_loc, a_loc);
return delta_cost;
#end_block

delta_cost -= a->old_cost;
delta_cost -= b->old_cost;
delta_cost += b->new_cost;
return delta_cost;
}

A.19 facesim

Facesim is a physics simulation application that models the human face and a time sequence of muscle activation. In the initialization phase of each time step, this application resizes several array structures using the value of extended_elements. If the value of extended_elements remains the same, the process of resizing arrays is redundant. Therefore, we define the m field in the LIST_ARRAYS class, where the value of extended_elements comes from, as the data trigger. The data trigger initiates the resizeArraysThread support thread function and allows us to skip the code in the Update_Position_Based_State_Parallel when possible.

We show our modification with the following code sections.

Data trigger declaration

class LIST_ARRAYS
{
  public:
    typedef T ELEMENT;

    ARRAYS<T> array;
int m; //trigger resizeArrayThread();

The support thread function

#DTT resizeArrays
void resizeArraysThread(int *m) {
int extended_elements=m;
threading_auxiliary_structures->
extended_U->Resize_Array(extended_elements);
threading_auxiliary_structures->
extended_De_inverse_hat->Resize_Array(extended_elements);
threading_auxiliary_structures->
extended_Fe_hat->Resize_Array(extended_elements);
threading_auxiliary_structures->
extended_dP_dFe->Resize_Array(extended_elements);
threading_auxiliary_structures->
extended_V->Resize_Array(extended_elements);
return;
}

The skippable region

Update_Position_Based_State_Parallel() {
    THREAD_POOL& pool=*THREAD_POOL::Singleton();
    #ifdef USE_REDUCTION_ROUTINES
    THREAD_DIVISION_PARAMETERS<T>&
    parameters=*THREAD_DIVISION_PARAMETERS<T>::Singleton();
    #endif

    LOG::Time("UPBS (FEM) - Initialize");
    #ifndef USE_REDUCTION_ROUTINES
    int extended_elements=
    threading_auxiliary_structures->extended_tetrahedrons->m;
    #block resizeArrays
    threading_auxiliary_structures->
    extended_U->Resize_Array(extended_elements);
    threading_auxiliary_structures->
    extended_De_inverse_hat->Resize_Array(extended_elements);
    threading_auxiliary_structures->
    extended_Fe_hat->Resize_Array(extended_elements);
    threading_auxiliary_structures->
    extended_V->Resize_Array(extended_elements);
    }
threading_auxiliary_structures->
extended_dP_dFe->Resize_Array(extended_elements);
threading_auxiliary_structures->
extended_V->Resize_Array(extended_elements);
#end_block
node_stiffness->
Resize_Array
(strain_measure.tetrahedralized_volume.particles.number);
threading_auxiliary_structures->
extended_edge_stiffness->
Resize_Array(threading_auxiliary_structures->extended_edges->m);

A.20 fluidanimate

Fluidanimate is a physics simulation application that generates data from surface rendering for computer animations. The algorithm in this application contains four steps when processing each frame – RebuildGrid, ComputeForces, ProcessCollisions, AdvanceParticles. Conventional parallelism partitions the computation within each step and places barriers between steps. However, we found that the computation in the ProcessCollisions function can start earlier – as soon as the required data is ready.

For our DTT implementation in this application, we modify the Cell data structure by adding an update flag. The flag is incremented every time the ComputeForces function updates the content of an element in the Cell array. By declaring the update field as the data trigger, the increment of the update flag causes the execution of the ProcessCollisionsThread function. Therefore, the DTT model starts this computation earlier than conventional parallelism but also avoids the synchronization overhead between these two steps.

The following code sections illustrate our changes.
Data trigger declaration

```c
struct Cell {
    Vec3 p[16];
    Vec3 hv[16];
    Vec3 v[16];
    Vec3 a[16];
    float density[16];
    int update; #trigger ProcessCollisionsThread();
};
```

The support thread function

```c
#DTT ProcessCollisions
void ProcessCollisionsThread(Cell *x) {
    const float parSize = 0.0002f;
    const float epsilon = 1e-10f;
    const float stiffness = 30000.f;
    const float damping = 128.f;
    int i = x - &cells[0];
    Cell &cell = cells[i];
    int np = cnumPars[i];
    for(int j = 0; j < np; ++j) {
        Vec3 pos = cell.p[j] + cell.hv[j] * timeStep;
        float diff = parSize - (pos.x - domainMin.x);
        if(diff > epsilon)
            cell.a[j].x += stiffness*diff - damping*cell.v[j].x;
        diff = parSize - (domainMax.x - pos.x);
        if(diff > epsilon)
            cell.a[j].x -= stiffness*diff + damping*cell.v[j].x;
        diff = parSize - (pos.y - domainMin.y);
        if(diff > epsilon)
            cell.a[j].y += stiffness*diff - damping*cell.v[j].y;
        diff = parSize - (domainMax.y - pos.y);
        if(diff > epsilon)
            cell.a[j].y -= stiffness*diff + damping*cell.v[j].y;
    }
}
```
diff = parSize - (pos.z - domainMin.z);
if(diff > epsilon)
cell.a[j].z += stiffness*diff - damping*cell.v[j].z;

diff = parSize - (domainMax.z - pos.z);
if(diff > epsilon)
cell.a[j].z -= stiffness*diff + damping*cell.v[j].z;
}
}

The skippable region

void AdvanceFrame()
{
    RebuildGrid();
    ComputeForces();
    #block ProcessCollisions
    ProcessCollisions();
    #end_block
    AdvanceParticles();
}

A.21 Swaptions

Swaptions is a financial benchmark that uses the Heath-Jarrow-Morton framework to price a portfolio of stock options. The benchmark spends most of its time in the worker method that walks through each element in the swaptions array and calls the HJM_Swaption_Blocking using each of them. The HJM_Swaption_Blocking function is an idempotent function because this function does not overwrite any of the inputs. In addition, our profiler also found that the inputs for each call are the same in this benchmark. Therefore, we create several global variables that store the inputs of the previous invocation of the HJM_Swaption_Blocking function. We only trigger the HJM_Swaption_Blocking_thread support thread function that calls the HJM_Swaption_Blocking function when the inputs change. We declare those global
variables as data triggers and mark the original code that calls the HJM_Swaption_Blocking function as the skippable region since the HJM_Swaption_Blocking_thread support thread function will replace the computation.

The following code sections contain our modifications.

**Data trigger declaration**

```
FTYPE dStrike_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE dCompounding_thread;
    #trigger HJM_Swaption_Blocking_thread();
FTYPE dMaturity_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE dTenor_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE dPaymentInterval_thread;
    #trigger HJM_Swaption_Blocking_thread();
int iN_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE dYears_thread; #trigger HJM_Swaption_Blocking_thread();
int iFactors_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE *pdYield_thread; #trigger HJM_Swaption_Blocking_thread();
FTYPE **ppdFactors_thread;
    #trigger HJM_Swaption_Blocking_thread();
```

**The support thread function**

```
#DTT HJM_Swaption_Blocking
void HJM_Swaption_Blocking_thread(void *x)
{
    HJM_Swaption_Blocking(pdSwaptionPrice, dStrike_thread,
                 dCompounding_thread, dMaturity_thread,
                 dTenor_thread, dPaymentInterval_thread,
                 iN_thread, iFactors_thread,
                 dYears_thread,
                 pdYield_thread, ppdFactors_thread,
                 100, NUM_TRIALS, BLOCK_SIZE, 0);
}
```

**The skippable region**

```
void * worker(void *arg){
```
int tid = *((int *)arg);

int chunksize = nSwaptions/nThreads;
int beg = tid*chunksize;
int end = (tid+1)*chunksize;
if(tid == nThreads -1 )
    end = nSwaptions;
for(int i=beg; i < end; i++) {
    Id_thread = swaptions[i].Id;
    dStrike_thread = swaptions[i].dStrike;
    dCompounding_thread = swaptions[i].dCompounding;
    dMaturity_thread = swaptions[i].dMaturity;
    dTenor_thread = swaptions[i].dTenor;
    old_value.d = dPaymentInterval_thread;
    dPaymentInterval_thread = swaptions[i].dPaymentInterval;
    iN_thread = swaptions[i].iN;
    dYears_thread = swaptions[i].dYears;
    iFactors_thread = swaptions[i].iFactors;
    pdYield_thread = swaptions[i].pdYield;
    ppdFactors_thread = swaptions[i].ppdFactors;
#block HJM_Swaption_Blocking
    int iSuccess = HJM_Swaption_Blocking
        (pdSwaptionPrice, swaptions[i].dStrike,
        swaptions[i].dCompounding,
        swaptions[i].dMaturity,
        swaptions[i].dTenor,
        swaptions[i].dPaymentInterval,
        swaptions[i].iN, swaptions[i].iFactors,
        swaptions[i].dYears,
        swaptions[i].pdYield, swaptions[i].ppdFactors,
        100, NUM_TRIALS, BLOCK_SIZE, 0);
    assert(iSuccess == 1);
#end_block
    swaptions[i].dSimSwaptionMeanPrice = pdSwaptionPrice[0];
    swaptions[i].dSimSwaptionStdError = pdSwaptionPrice[1];
}
return NULL;

A.22 vips

The vips benchmark is derived from the VASARI Image Processing System. It transforms the image through 18 different pipeline stages. Our profiler suggests that
one of the pipeline stages that uses the `im_lintra_vec` function with `IMAGE t[12]` as the input creates redundant computation on vectors with zeros. We also found that the computation of this stage can be triggered in parallel earlier if we rearrange the pipeline somewhat to generate the value of `t[12]` earlier.

In our DTT version of the code, we declare an `IMAGE` variable `t12` to replace the `t[12]` and use this variable as the data trigger. The content in this variable can be produced after we know the value of `t[4]`. So we move the generation of `t12` right after the pipeline stage that generates `t[4]`. Once the content in `t12` changes, the DTT model can immediately execute the support thread function — `im_lintra_vec_zero_thread` or avoid redundant computation if `t12` remains the same. The `im_lintra_vec_zero_thread` function is also a specialized version of the `im_lintra_vec` function given that one of the input images contains only zeros to further reduce the amount of computation.

We show our DTT implementation as the following.

**Data trigger declaration**

```c
IMAGE *t12; #trigger im_lintra_vec_zero_thread();
```

**The support thread function**

```c
#DTT im_lintra_vec_zero_thread
int
im_lintra_vec_zero_thread( IMAGE *in)
{
    LintraInfo *inf;
    int i;
    double *b = white;
    double *a = zero;
    IMAGE *out = t[13];
    /* Check args.
     */
    if( in->Coding != IM_CODING_NONE ) {
```
im_error( "im_lintra_vec", _( "not uncoded" ) );
return( -1 );
}

if( n != in->Bands && (n != 1 && in->Bands != 1) ) {
im_error( "im_lintra_vec",
_( "not 1 or %d elements in vector" ), in->Bands );
return( -1 );
}

/* Prepare output header. */
if( im_cp_desc( out, in ) )
return( -1 );
if( im_isint( in ) ) {
out->Bbits = IM_BBITS_FLOAT;
out->BandFmt = IM_BANDFMT_FLOAT;
}
if( in->Bands == 1 )
out->Bands = n;

/* Make space for a little buffer. */
if( !(inf = IM_NEW( out, LintraInfo )) ||
!(inf->a = IM_ARRAY( out, n, double )) ||
!(inf->b = IM_ARRAY( out, n, double )) )
return( -1 );
inf->n = n;
for( i = 0; i < n; i++ ) {
inf->a[i] = 0;
inf->b[i] = white[i];
}
/* Generate! */
if( n == 1 ) {
if( im_wrapone( in, out,
(im_wrapone_fn) lintra1_gen_zero, in, inf ) )
return( -1 );
}
else if( in->Bands == 1 ) {
if( im_wrapone( in, out,
(im_wrapone_fn) lintranv_gen_zero, in, inf ) )
return( -1 );
}
else {
if( im_wrapone( in, out, (im_wrapone_fn) lintran_gen_zero, in, inf ) )
return( -1 );
return( 0 );
}

The skippable region

im_extract_area( t[0], t[1], 100, 100, t[0] -> Xsize - 200, t[0] -> Ysize - 200 );
im_affine( t[1], t[2], 0.9, 0, 0, 0.9, 0, 0, 0, t[1] -> Xsize * 0.9, t[1] -> Ysize * 0.9 );
im_extract_band( t[2], t[3], 0 );
im_moreconst( t[3], t[4], 99 );
im_black( t12, t[4] -> Xsize, t[4] -> Ysize, 3 );
#block im_lintra_vec_zero
   im_lintra_vec( 3, zero, t12, white, t[13] );
#end_block
   read(pfm_fd[0], values, sizeof(values));
   totalDTTCycle +=(values[0]-PFM_DTT_start);
return(
   im_lintra_vec_zero_1( 3, darken, t[2], zero, t[5] ) ||
   im_Lab2XYZ( t[5], t[6] ) ||
   im_recomb( t[6], t[7], d652d50 ) ||
   im_lintra_vec_zero_1( 3, whitepoint, t[7], zero, t[8] ) ||
   im_lintra( 1.5, t[8], 0.0, t[9] ) ||
   im_XYZ2Lab( t[9], t[10] ) ||
   im_lintra_vec( 3, one, t[10], shadow, t[11] ) ||
   im_ifthenelse( t[4], t[13], t[11], t[14] ) ||
   im_Lab2LabQ( t[14], t[15] ) ||
   im_sharpen( t[15], out, 11, 2.5, 40, 20, 0.5, 1.5 ) );

A.23 x264

X264 is a video codec application. Our profiling tool found that the application
 can generate redundant computation in the process of setting up a frame context if the
 i_type field of the h->fenc remains the same. Therefore, our implementation stores the
previously used i_type field in the i_type_current_frame. We update this variable when the program moves to the next frame. We declare this variable as a data trigger and only trigger the computation depending on the i_type field if necessary.

We summarize our code modifications in the following code snippets.

Data trigger declaration

```c
int i_type_DTT; #trigger x264_slice_init_thread();
```

The support thread function

```c
#DTT frameInit
void *x264_slice_init_thread(void *x)
{
    int i_nal_type;
    int i_nal_ref_idc;
    int i_global_qp;

    if( i_type_DTT == X264_TYPE_IDR )
    {
        /* reset ref pictures */
        x264_reference_reset( h );

        i_nal_type = NAL_SLICE_IDR;
        i_nal_ref_idc = NAL_PRIORITY_HIGHEST;
        h->sh.i_type = SLICE_TYPE_I;
    }
    else if( i_type_DTT == X264_TYPE_I )
    {
        i_nal_type = NAL_SLICE;
        i_nal_ref_idc = NAL_PRIORITY_HIGH;
        h->sh.i_type = SLICE_TYPE_I;
    }
    else if( i_type_DTT == X264_TYPE_P )
    {
        i_nal_type = NAL_SLICE;
        i_nal_ref_idc = NAL_PRIORITY_HIGH;
        h->sh.i_type = SLICE_TYPE_P;
    }
    else if( i_type_DTT == X264_TYPE_BREF )
    {
```
i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_HIGH;
h->sh.i_type = SLICE_TYPE_B;
}
else /* B frame */
{
    i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_DISPOSABLE;
h->sh.i_type = SLICE_TYPE_B;
}

The skippable region in the x264_encoder_encode function

/* 4: get picture to encode */
if( h->fenc == NULL )
{
    /* waiting for filling bframe buffer */
    h->fenc = x264_frame_shift( h->frames.current );
    pic_out->i_type = X264_TYPE_AUTO;
    return 0;
}
i_type_DTT = h->fenc->i_type;

i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_HIGHEST;
h->sh.i_type = SLICE_TYPE_I;
else if( h->fenc->i_type == X264_TYPE_I )
{

    /* reset ref pictures */
    x264_reference_reset( h );

    }
i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_HIGH;
h->sh.i_type = SLICE_TYPE_I;
}
else if( h->fenc->i_type == X264_TYPE_P )
{
  i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_HIGH;
h->sh.i_type = SLICE_TYPE_P;
}
else if( h->fenc->i_type == X264_TYPE_BREF )
{
  i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_HIGH;
h->sh.i_type = SLICE_TYPE_B;
}
else /* B frame */
{
  i_nal_type = NAL_SLICE;
i_nal_ref_idc = NAL_PRIORITY_DISPOSABLE;
h->sh.i_type = SLICE_TYPE_B;
}

#end_block
h->fdec->i_poc =
h->fenc->i_poc = 2 * (h->fenc->i_frame - h->frames.i_last_idr);

h->fdec->i_type = h->fenc->i_type;
h->fdec->i_frame = h->fenc->i_frame;
h->fenc->b_kept_as_ref =
  h->fdec->b_kept_as_ref = i_nal_ref_idc != NAL_PRIORITY_DISPOSABLE && h->param.i_keyint_max > 1;
Bibliography


