CDTT: Compiler-Generated Data-Triggered Threads

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

Abstract

This paper presents CDTT, a compiler framework that takes C/C++ code and automatically generates a binary that eliminates dynamically redundant code without programmer intervention. It does so by exploiting underlying hardware or software support for the data-triggered threads (DTT) programming and execution model. 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.

1. Introduction

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 *statically redundant* code with techniques such as dead code elimination. Techniques to remove *dynamically redundant* code have been proposed, (e.g., automatic memoization [16, 19]), but the cost of checking redundance of the inputs scales with the size of the inputs themselves. This paper 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.

This optimization leverages the data-triggered threads (DTT) programming and execution model [21], which has been shown 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 work shows that the effect of eliminating unnecessary, dynamically redundant computation can be especially powerful [21].

Prior research on the DTT model [21, 22, 23] 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.

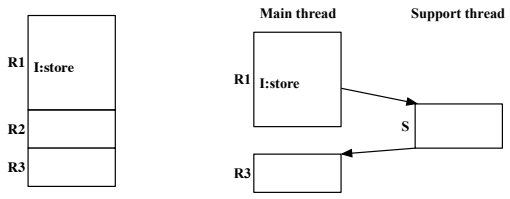
In this paper, we design a compiler framework, CDTT (compiler-generated data-triggered threads), which automatically generates data-triggered threads from applications written in C/C++, without requiring any modification to the source code. Because a support thread function can execute multiple times (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 [8].

Since DTT achieves significant performance improvement through eliminating redundant computation, we also present profile-assisted CDTT, which can use profile-generated information about redundance 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 [15, 5]), it typically does so with relatively low overhead.

CDTT can work with either hardware support for DTT [21] or the software-only implementation [23]. To maximize generality, in this paper we demonstrate CDTT

978-1-4799-3097-5/14/$31.00



(a) The original application (b) DTT execution model

Figure 1. The data-triggered thread execution model

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 [12]. The compiled applications use a DTT runtime system similar to the software DTT infrastructure described in [23]. 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.

This paper makes several contributions. It demonstrates that the automatically generated data-triggered threads can achieve nearly the same level of performance improvement as careful human coding. It describes an analysis framework that identifies potential code regions for applying data-triggered threads. It presents algorithms for automatically composing support thread functions. It demonstrates that the algorithms for selecting regions for DTT formation also serve as an accurate static predictor of redundant computation.

This paper is organized as follows. Section 2 provides an overview of the DTT model, the underlying runtime system, and the algorithm for detecting idempotence. Section 3 discusses other related work. Section 4 details the design of our compilation framework. Section 5 describes the design of profile-assisted CDTT, which specifically targets redundant computation. Section 6 presents our experimental methodology. Section 7 discusses the experimental results. Section 8 concludes.

2. Background

This section provides some background on the data-triggered threads model that our compiler targets. We also discuss how to support data-triggered threads using hardware or a pure software runtime system running on an off-the-shelf processor. Finally, we provide a short description of how we detect the idempotent property of selected code regions.

2.1. Data-triggered threads model

Unlike conventional execution models that initiate parallelism using control flow, the data-triggered threads [21, 22] model initiates parallelism when the application generates a new value. This carries two primary benefits. First, this mechanism exposes parallelism, and in particular does so immediately after the generation of source data. Second, if the data is not changed, the program does not perform the computation. Using the DTT model, the programmer composes support threads that execute only when the data actually changes.

In this way, DTT makes it natural to express computation in such a way that it is only performed when necessary. That paper give the example of a matrix addition that is typically coded by sweeping through the full arrays, regardless of how many cells in the arrays have changed since the last matrix addition. DTT easily expresses this computation such that only the cells whose inputs have changed are recalculated. The original DTT paper demonstrated that DTT is especially effective in avoiding redundant computation to achieve significant speedup in some applications [21].

Assume that an application contains code regions R1, R2, and R3, where the computation of R2 depends on the memory output of the I:store instruction in R1 shown in Figure 1(a). A conventional execution model would execute the computation in R2 each time through this code. With the DTT model (Figure 1(b)), the application can execute the computation in R2 in parallel using a support thread S immediately after I:store produces new data inputs. The programmer can either use the support thread S to perform the same dependent computation or implement an incremental version of R2. After the support thread S completes, the DTT model can skip the execution of R2 and continue executing R3 since the support thread S already performed the computation of R2. The DTT model calls the location that I:store modifies a *data trigger*. Any modification of a data trigger will spawn the execution of a support thread. The beginning of the original code region R2 (the end of code region R1) is an implicit barrier in the DTT model because the runtime will stall the main thread to prevent interference between the main thread and the support thread if the main thread reaches the implicit barrier before S completes, since it makes no assumptions about dependences between R2 and R3.

If I:store is the only instruction that can affect the input of R2 and I:store does not change memory or does not execute, the input data of R2 remains the same and the previously executed result of R2 is still valid in memory. To avoid redundant computation in this case, the DTT model will not spawn a new support thread, but will skip the execution of R2. The DTT model calls the code region R2 a *skippable region*. Because the DTT model can fail to spawn a support thread or disable the support thread during runtime, the DTT model still keeps the skippable region R2 in the main thread to be executed in those (typically rare) cases.

Prior DTT research [21, 22, 23] has required programmers to modify or rewrite applications. Programmers need to add pragmas to identify data triggers, create support thread functions, and annotate the skippable regions. In

contrast, CDTT allows unmodified legacy code to use the DTT model.

2.2. Supporting data-triggered threads execution

Prior work demonstrates both (1) a hardware and instruction set architecture based approach [21, 22] and (2) a software-only approach which runs on existing hardware [23] to provide the required support for DTT. Although our work takes the approach of the latter, we discuss both solutions in the following paragraphs because CDTT will work in either case.

To support data-triggered threads in hardware [21, 22], the processor contains additional hardware tables – the *thread registry*, the *thread queue*, and the *thread status table*. The thread registry records the static information of support thread functions and the corresponding skippable regions. The thread queue provides a temporary storage for support thread events that have not completed. The thread status table keeps track of the runtime information regarding each skippable region (is it skippable/valid, is it invalid and needs to execute, are there still support threads outstanding/pending?).

The hardware solution also requires several new instructions. The **tstore** instruction is the most important one among these instructions. It detects whether the store changes the existing memory content and should trigger a support thread event. The compiler uses **tstore** instructions instead of traditional memory store instructions for writes to data triggers. This mechanism allows the system to track changes for arbitrarily large data structures and skip any size code regions without significant storage overhead. Previous works that also focus on eliminating redundant computation [3, 11, 15, 5] typically need storage that scales with the data size.

We can also support the DTT model on existing hardware using a software-only approach [23]. The software replaces the functionality of hardware tables in the runtime system using two global data structures, the *thread queue* and *state variables*. The software thread queue provides storage for pending support thread events. For each skippable region, the runtime system allocates a state variable associated with it. The state variable contains information that indicates whether the runtime system can skip the execution of the skippable region.

The runtime library of the software-only approach also provides functions to replace the additional instructions (e.g. the **tstore** instruction) used in the hardware solution. The runtime library also provides a **dtt_barrier** function to serve as the implicit barrier before each skippable region.

With no hardware support, the software-only approach results in some runtime overhead. The function call overhead for detecting memory content changes and managing the global data structures is relatively small if the programmer only targets very few variables or data structure fields. To minimize the thread spawning overhead, the DTT runtime system creates one polling thread at the beginning of execution. The polling thread monitors the TQ and executes support thread events from the TQ.

```
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 2. An example of idempotent code

The DTT model can potentially degrade application performance and cause the main thread to stall if the code triggers a large number of support threads or does not provide sufficient slack to hide the support thread execution latency. To mitigate this problem, the software-only approach can incorporate a thresholding mechanism [23], which records the number of main-thread stalls before each skippable region. If this reaches a threshold during an interval, the system will stop spawning support threads for the corresponding skippable region, and simply invalidate the skippable region when the runtime system detects a data change. The system will thus execute the skippable region code instead of using support threads, thus returning to conventional execution. The system will periodically retry using the support threads.

2.3. Identifying idempotent regions

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 prior work, it is up to the programmer to create idempotent code. For this work, we need the ability to detect idempotence automatically, so we discuss those techniques here.

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 2 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 [8]. 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 there exists any data dependency that is a write to the register and 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.

3. Other related work

Memoization [15, 5, 16, 19] is a technique that reduces redundant computation by employing additional memory to cache the input and output values of instructions, blocks, or functions. Memoization techniques rely on the programmer to revise the code and algorithms [15, 5] or this can be done automatically by the compiler [16, 19]. The DTT model differs in that it does not need additional storage for data inputs and outputs and thus can optimize arbitrarily sized code regions and arbitrarily sized data structures.

There are also several compiler optimizations that avoid redundant computation by eliminating some redundant load-reuse cases [3, 11]. However, these works can only optimize relatively small code regions. These techniques, similarly, cannot be applied to code with large memory footprints.

Dataflow architectures trigger execution upon the generation of data and achieve fine-grain parallelism implicitly [20, 1, 17, 6]. Pure dataflow programming languages [4, 18] allow programmers to describe programs as dataflow graphs. These languages differ significantly from imperative programming languages and hardware support for dataflow can be complex.

Cilk [9], CEAL [10], and DTT [21] extend the C/C++ programming language to support dataflow-like programming and execution models on conventional architectures. Cilk allows the programmer to exploit parallelism similar to the dataflow model. CEAL facilitates the use of incremental algorithms on changing data to avoid redundant computation. DTT benefits from exploiting dataflow-like parallelism like Cilk, but also avoids redundant computation as CEAL. However, the above language extensions still place all the burden on the programmer when writing programs.

CDTT creates dataflow-like concurrency on top of imperative program languages, similar to Program Demultiplexing (PD) [2]. Both CDTT and PD perform code analysis for existing programs and try to produce code that triggers parallelism as soon as the program produces the inputs of a code region. However, PD executes functions or methods speculatively and 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.

de Kruijf et al. [8, 7] take advantage of idempotence to support low-overhead fault recovery. CDTT uses idempotence in identifying potential data triggered threads. 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.

4. 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 [12]. 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.

4.1. 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.

(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.

CDTT then tests each region in the candidate list for idempotence, using the mechanism described in Section 2.3. 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. 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, CDTT also removes the region from the list.

For example, assume that we have a program as shown in Figure 3. 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.

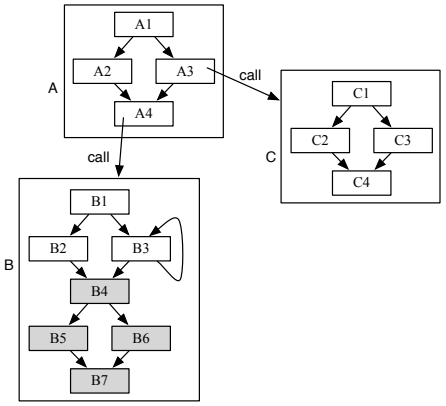


Figure 3. An example inter-procedural control flow graph

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 WAW dependence for these memory objects in the reachable code.

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.

4.2. 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 the 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 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 *running*. At the end of the support thread function, CDTT marks the state variable as *valid* or *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 **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.

4.3. 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 4.1. Because CDTT transforms all the inputs of a support thread function into global variables or memory addresses (Section 4.2), 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 **tstore**s 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 **tstore** function will trigger a support thread event even if the memory content that the last **tstore** compares remains the same — this is a slight change to the implementation of **tstore** from prior work. 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 *ammp* code from Figure 2 as an example). In these cases, CDTT replaces all the **tstore** function calls with **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.

5. Profile-assisted CDTT

The previous work on DTT [21] shows that using support threads to eliminate redundant computation brings the most significant performance gain. Therefore, we also present profile-assisted CDTT, in which the compiler focuses on identifying redundant computation based on input from the profiler. Profile-assisted CDTT inherits the framework of regular CDTT, but can accept profile data as an input. In this section, we will describe the generation of profile data and how profile-assisted CDTT selects DTT regions.

Although our results show only small benefit from profiling, we include this discussion and these results for three reasons. First, it allows us to demonstrate the unexpected result that the non-profile results nearly match the profile-based results. Second, there may still be some cases, outside of our application set where profiling is necessary. Third, in future implementations, where the compiler is able to create idempotence (as opposed to the current system that only detects idempotence), it is likely that profile information on what code should be extracted and placed in a new idempotent region would be of high value.

5.1. Profiling for redundant computation

The original DTT work [21] shows that 78% of loads in the SPEC2000 benchmarks are redundant – that is, they load data that (at one time) was previously loaded by this same load, and has not changed value since the previous

instance. The load, and typically the computation that depends on that load, is redundant and unnecessary. That work demonstrates significant performance improvement from eliminating redundant computation by using the profiling result for redundant loads. However, we found that data misleading in a few cases – a programmer could identify those cases, but our automatic system would have more difficulty. This would happen because in many cases there is a string of computation that depends on multiple loads but results in a single store. Identifying a single redundant load, therefore, does not always imply a redundant string of computation.

Therefore, in this work, we instead profile silent stores – store instructions that write the same value that is already stored at the target memory address [13]. If an instruction writes the same value to the same memory address, it is very likely the instruction stream that leads to the silent store is redundant. While Lepak and Lipasti [14] target skipping the store instructions, we seek to skip the whole computation string that leads to redundant stores, as in [21].

To explore the incidence of redundant computation in applications, we use LLVM [12] to implement a profiling tool for silent stores. The tool takes LLVM bitcode files as inputs and instruments the memory store instructions to compare the written value with the current memory contents and keeps track of the number of silent stores. We execute the instrumented code on the LLVM virtual machine and collect the profile data. When we profile the applications, we use the train dataset for SPEC2000 benchmarks.

Our profiling results show that the percentage of redundant stores varies for each application, ranging from 0.3% to 57%, with an average of 24%. For mcf, mesa, and vortex, where the previous work demonstrates significant redundant computation, we still see that more than half the store instances are silent. Though the percentage of silent stores in other benchmarks is not high, we still can find individual static store instructions that are silent most of the time, even for ammp, which contains only 0.3% silent stores. Table 2 in the result section gives a more detailed breakdown of silent store frequencies.

5.2. Identify redundant regions

Profile-assisted CDTT uses the same framework as regular CDTT except that it accepts profile data and a *silent store cutoff* to guide the selection of DTT regions. Assuming we have the profile data described, profile-assisted CDTT can associate each store from the profile data with a memory store in the LLVM bitcode. The profile data tells us the percentage of time that store was silent. We will use a *silent store cutoff* to statically mark each store as silent or not. If a store instruction contains a higher ratio of silent stores than the silent store cutoff, profile-assisted CDTT considers this instruction as silent.

For code regions that pass the idempotence and name dependence analysis, they now go through a redundance test. Any region that contains a single store that is not deemed silent is eliminated from consideration.

6. 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 prior work. 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 paper, 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 paper. We use a DTT runtime system without the thresholding mechanism [23] (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.

7. 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.

7.1. 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 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 the numbers for applications compiled with silent store cutoffs of

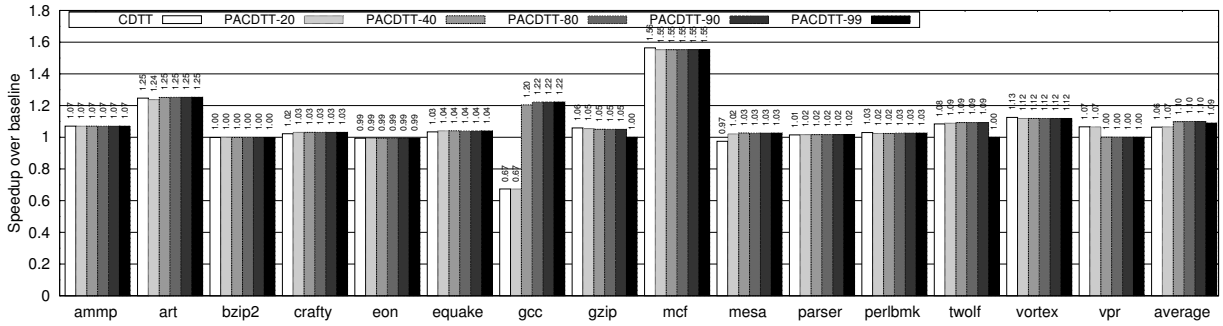


Figure 4. The speedup of CDTT by using different silent store cutoffs running on the multi-core runtime system

	Support thread functions			Average support thread instruction counts		
Name	80%	20%	0%	80%	20%	0%
ammp	1	1	1	24	24	24
art	1	3	3	24	46	46
bzip2	1	1	1	26	26	26
crafty	2	2	3	40	40	69
eon	1	1	1	24	24	24
equake	1	1	2	22	22	75
gcc	3	19	19	712	116	126
gzip	1	2	2	23	34	34
mcf	1	1	2	154	154	108
mesa	1	2	3	23	91	75
parser	1	1	2	24	24	50
perlbmk	4	4	5	37	49	54
twolf	3	5	5	34	28	28
vortex	2	2	3	13	13	22
vpr	0	1	3	0	26	40

Table 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%)

80%, 20%, and 0%. 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 4 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 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 7.2.

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 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 [23] 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 com-

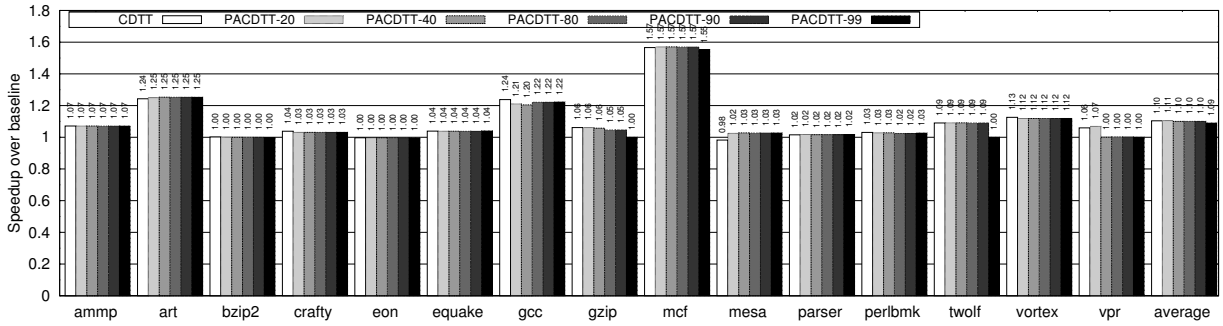


Figure 5. The speedup of profile-assisted CDTT with different silent store cutoffs running on the runtime system with thresholding

plete. Figure 5 presents the application performance when profile-assisted CDTT uses the DTT runtime system with thresholding, targeting different silent store cutoffs.

Relative to the prior work, we find that thresholding is actually less important for the automatically generated threads using profile-assisted CDTT than for programmer-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.

7.2. Discussion of CDTT without profiling

Perhaps most interesting, however, are the results in Figure 5 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 redundance of computation.

Table 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.

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

Table 2. The percentage of silent stores in DTT regions selected by CDTT (without profiling), compared to the percentage of silent stores in all 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 redundance. Therefore, in many cases, the carefully collected profile data serves only to confirm the identification of the redundant regions.

7.3. Redundance vs. parallelism in CDTT

Like prior applications of DTT, CDTT exploits both redundance and parallelism. In this section, we perform experiments that allow us to separate the effects.

Figure 6 compares the performance result of (1) running

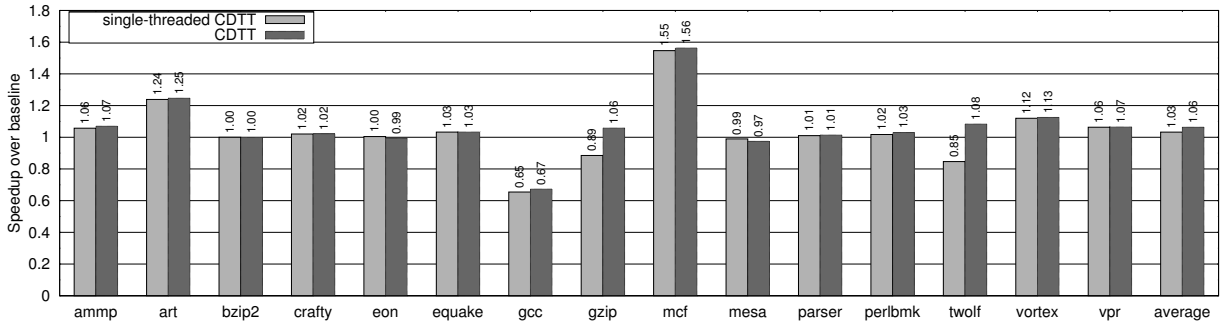


Figure 6. The speedup of CDTT on single-threaded and multi-core runtime system

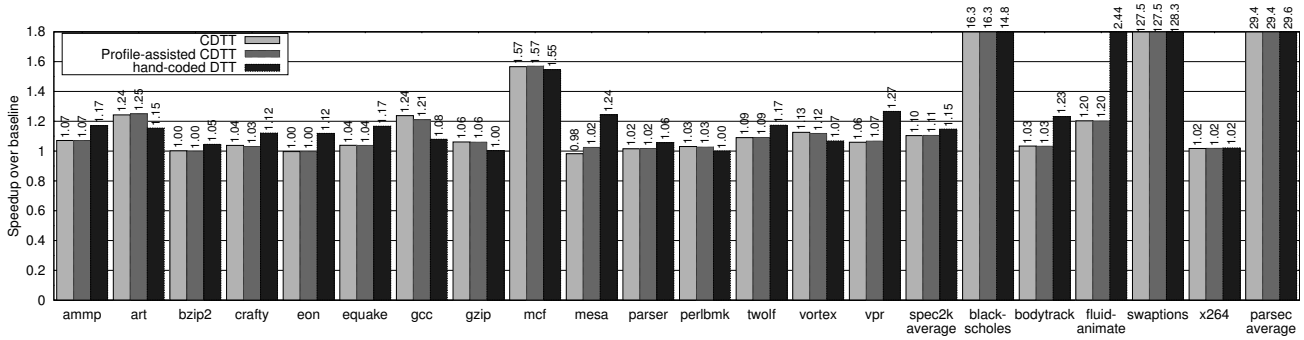


Figure 7. The speedup of CDTT without profile (CDTT), profile-assisted CDTT with 20% silent store cutoff (profile-assisted CDTT) and programmers' modification (hand-coded DTT) running on multi-core runtime system with thresholding

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 redundance 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 redundance. 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 redundance 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.

7.4. CDTT and hand-coded DTT

Figure 7 compares our automatically generated results with the carefully hand-coded results from prior work [23]. 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, the programmer in previous DTT works 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 redundance. 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.

The prior work also showed very high speedups for the PARSEC benchmarks (e.g., 16X on one benchmark and 128X on another). In this work, we have not shown those results in earlier graphs. This is because, while they show true redundant behavior, 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. Thus, the PARSEC results are also included in Figure 7. 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 simlarge dataset to collect profile data and use the native dataset to measure the performance. The experimental result in Figure 7 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.

7.5. 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.

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 benchmarks 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 number 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 [21]. In that case, nearly all of the measured overhead will go away. This implies that our non-profiled results could improve significantly (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.

8. Conclusions

This paper 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 this work is that idempotence and name dependence analysis becomes a highly effective static filter for redundant code identification, rendering profiling unnecessary.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was funded in part by NSF grants CCF-1018356 and CCF-1219059.

References

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

[2] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *33rd Annual International Symposium on Computer Architecture*, pages 302–313, June 2006.

[3] R. Bodík, R. Gupta, and M. L. Soffa. Load-reuse analysis: design and evaluation. In *ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 64–76, May 1999.

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

[5] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multimedia processing by implementing memoing in multiplication and division units. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–261, October 1998.

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

[7] M. de Kruijf and K. Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *IEEE/ACM 2013 International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, 2013.

[8] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *ACM SIGPLAN 2012 conference on Programming Language Design and Implementation*, pages 475–486, June 2012.

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

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

[11] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C. cheow Lim, J. Ng, and D. Sehr. An advanced optimizer for the IA-64 architecture. *IEEE Micro*, 20:60–68, 2000.

[12] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2004 International Symposium on Code Generation and Optimization*, Mar 2004.

[13] K. Lepak and M. Lipasti. On the value locality of store instructions. In *27th Annual International Symposium on Computer Architecture*, pages 182–191, March 2000.

[14] K. Lepak and M. Lipasti. Silent stores for free. In *33rd International Symposium on Microarchitecture*, pages 22–31, December 2000.

[15] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[16] J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *9th International Joint Conference on Artificial intelligence*, pages 165–172, 1985.

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

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

[19] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, March 1991.

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

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

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

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