In Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA 2011)

Data-Triggered Threads: Eliminating Redundant Computation

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 introduces the concept of data-triggered threads. Unlike threads in parallel programs in conventional programming models, these threads are initiated on a change to a memory location. This enables increased parallelism and the elimination of redundant, unnecessary computation. This paper focuses primarily on the latter.

It is shown that 78% of all loads fetch redundant data, leading to a high incidence of redundant computation. By expressing computation through data-triggered threads, 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%.

1. Introduction

Any von Neumann-based architecture that exploits parallelism does so by initiating computation (i.e., threads) based on the program counter. Typically, it is the program counter reaching a *fork* or maybe a *pthread_create* call. Even for helper thread architectures [7, 8, 30, 31] or thread level speculation [20, 25, 27], it is the program counter reaching a trigger instruction or a spawn point. In dataflow architectures [3, 10], however, computation is initiated when data is written. This has two key advantages – parallelism is maximized because the initiated code is available for execution immediately rather than when the program counter reaches the dependent code, and unnecessary computation (when data is not changed) is not executed. However, dataflow architectures have not yet proven to be commercially viable due to the complexity of the token matching, the communication overheads of fine-grain parallelism, and asynchronous instruction triggering.

This research proposes a new programming and execution paradigm that enables these two characteristics of dataflow architectures through minor architectural changes to a conventional von Neumann architecture. We call this paradigm *data-triggered threads*. In this model, a thread of computation is initiated when an address is touched or changed. Similar to the dataflow architecture, the dependent code is available for execution immediately, regardless of the position of the program counter. More importantly, at least in our initial experiments, data that is not changed never spawns unnecessary computation. This can result in dramatic performance and energy gains. However, because thread generation only depends on changes to a single address, we completely bypass the complex token-matching mechanisms required to make true dataflow work.

This paper, as an initial look at this programming and execution model, focuses primarily on the opportunity of redundant computation. 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. We use our analysis of redundant computation to guide us in making relatively minor changes to some of these benchmarks, exploiting this new execution model. Speedups vary, but range as high as 5.89, and average 1.46.

Consider a simple example of code that operates on two arrays A and B, then at the end of each iteration computes C as the matrix product of A and B. Typically, we completely recalculate every element of C, even though A and B may have only changed slightly, or even not changed at all. In our model, you can specify that C, or specific elements of C, are recalculated as soon as A or B are changed. In essence, we are specifying invariants – C will always be (modulo the time to execute the thread) the product of A and B.

Since the threads spawned in the data-triggered architecture are non-speculative threads, we do not require additional hardware to keep track of different versions of data or squash speculative threads, as in speculative multithreading architectures. Prior work, such as value prediction [19], dynamic instruction reuse [26], and silent stores [18] also exploit redundant computation, but typically short-circuit a single instruction or small block of instructions [12]. However, we can completely skip arbitrarily large blocks of

computation, including computation that reads and writes memory.

This paper makes the following contributions: (1) It shows that applications in general demonstrate very high rates of redundant loads, resulting in significant amounts of redundant computation. (2) It proposes a new programming model and execution paradigm — data-triggered threads. (3) It describes very modest architectural support to enable this execution model. (4) It shows that existing, complex code can be transformed to exploit data-triggered threads with almost trivial changes. Performance gains from these transformations can be very high.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 exposes the opportunity of redundant computation. Section 4 illustrates the execution model of data triggered threads. Section 5 describes the full programming model in more detail. Section 6 details the architectural changes to support the data-triggered thread programming model. Section 7 describes our experimental methodology. Section 8 presents and discusses the experimental results.

2. Related work

In contrast to the von Neumann model, instruction scheduling in dataflow models [3, 10] is only based on operand availability, and execution is not constrained by program sequencing. In addition, instructions with their operands available can be processed concurrently to achieve fine-grain parallelism. However, the classic dataflow processors are hampered by the hardware complexity.

To provide a smoother transitions from von Neumann to dataflow architectures, hybrid architectures [9, 14, 23], StarT [2, 22], EARTH [13], DDM [11, 16], and Fuce [1] attempt to build dataflow machines on top of conventional architectures. However, these systems still require significant changes to a baseline architecture to better support message passing, thread management, synchronization, context switching, and memory accesses.

A number of prior proposals have exploited redundant computation. Value prediction [19] predicts the output of an instruction based on previous execution results stored in the prediction table, and executes the instruction using the predicted values speculatively. Dynamic instruction reuse [26] buffers the execution result of instructions, but skips the execution stage of reused instructions (whose inputs match saved prior invocations). Thus, each instruction is reused non-speculatively as long as the inputs match. Block reuse [12] expands the reuse granularity to a basic block. Silent stores [17, 18] detects and removes store instructions from the load/store queue by exploiting the free read ports of the cache. However, all of 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. Data-triggered threads do not share this limitation. In addition, our technique never needs to recover from misspeculation on data values.

Compiler optimizations [5, 15] can eliminate some redundant load-reuse cases. However, they work on relatively small blocks of code, and on load instructions with a relatively small memory footprint.

Memoization [6, 21] 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 additional storage (to hold the values of all inputs) and significant change in algorithms. Conversely, DTT 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.

Program Demultiplexing (PD) [4] shares the goal of triggering threads as soon as the program produces the appropriate values. However, PD speculatively executes functions or methods, and requires additional hardware to buffer the speculative results. In addition, PD does not have the ability to skip redundant computation, so PD never decreases instruction count.

3. Redundant loads

In a simplified model of execution, a program can be considered as composed of many strings of computation, often beginning with one or more loads of new data, followed by computation on that data using registers, and completing with one or more stores to memory. If the data loaded has not changed since the previous invocation of this code, it is likely the computation produces the same results, and the same values get written to the same locations in memory. The last effect was demonstrated by Lepak and Lipasti [17], where they demonstrated that 20-68% of all stores are silent. We are more interested in the other end, because we want to skip the entire string of computation, not just the store. Therefore, we study the incidence of *redundant loads*. A redundant load is one where the last time this load loaded this address, it fetched the same value (there may have been intervening accesses to other addresses, so this is not necessarily a redundancy that even value prediction would catch).

Figure 1 shows the results of this experiment on the

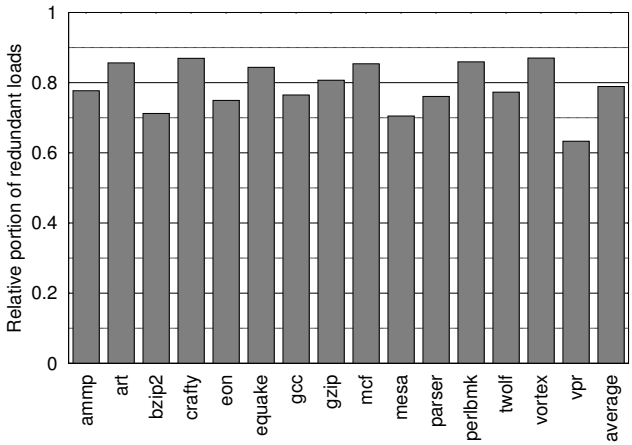

Figure 1. The redundant load instructions in our tested benchmarks

```
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;
         checksum++;
    }
    tmp = node;
    node = node->child;
}
```

Figure 2. Source code segment from `refresh_potential` function of mcf

SPEC2000 C benchmarks. 78% of all loads are redundant. Nearly all executed instructions depend (directly or indirectly) on at least one load and thus inherit much of that redundancy. Further, our measurements show that over 50% of all instructions are redundant (depend only on redundant loads).

Figure 2 shows an example of redundant behavior in mcf. The while loop in the `refresh_potential` function updates the `potential` of all nodes in a network based on the value of the `potential` and `cost` fields of linked 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 changes.

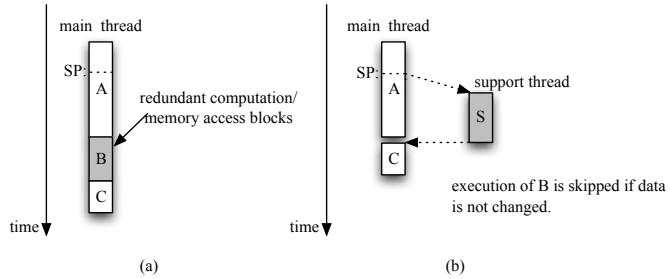

Figure 3. Execution model of data-triggered thread architecture

4. Overview of data triggered threads

Figure 3 shows the basic operation of the data-triggered thread execution model. The original application contains code sections A, B, and C. The execution of code section B depends on data generated by code section A. If the instruction SP in code section A generates a value different from what is currently stored in memory, the system will spawn a support thread (S) in a free hardware context. The spawned thread performs the computation of section B in non-speculative fashion.

After the support thread (S) 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 S with exactly the same input data. If the instruction SP does not modify the value stored in memory, the computation and memory access operations of section B are redundant. In this case, the data-triggered thread architecture will not spawn a thread but will skip the instructions in section B to eliminate the redundant computation – the computation B last did with these inputs is still valid.

Take the code in Figure 2 as an example — `potential` only depends on `potential` of this node and `cost` and `potential` of linked nodes (via the `pred` and `basic_arc` edges). In other words, the value of `potential` changes only when the program changes the value of `potential` or `cost` of of its `pred` and `basic_arc` nodes, or when the node has different `pred` and `basic_arc` nodes. In the data-triggered thread model, assignments which change `potential`, `cost` or the node tree structure will trigger a support thread performing the computation to update `potential` of the subtree that is rooted at the changed node. When the main thread reaches the original `refresh_potential` function, our model will skip the execution of that function, because all of the values that need to change will have already been computed and written.

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.

Clearly, the code in B must have certain properties, which will be discussed in the next section. This research presents a host of possible policy decisions which impact both the level of architectural support required and the programming model. The particular decisions we made for this paper are discussed in Section 6, but many of those decisions will be open to re-evaluation in future work. As one example, we favor very coarse-grain threads – in the matrix multiply example, this would mean that instead of recalculating just the minimum number of cells of the result matrix whenever a source cell is changed, we would do the entire matrix multiply when one of the source matrices change. This only pays off when the entire array rarely changes, but greatly reduces the amount of internal storage for pre-executed threads. In most cases, we are only tracking a few potential data-triggered threads.

5. Programming model

We have designed the data-triggered thread programming model to be a simple extension to conventional imperative programming languages. In particular, for these experiments, we have designed it to be easily implemented with annotations processed by a C program preprocessor.

To create a data-triggered thread, the programmer must identify specific computation that meets the requirements for a data-triggered thread, specify the data trigger, and generate the code for the data-triggered thread.

5.1. 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.

Data triggered threads return a single result (like any other function), although they can cause side effects in memory (updating heap structures, etc.).

One new constraint that C programmers may not be used to is that each data-triggered thread must be *restartable*. 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 2, 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 data-triggered threads that are not restartable.

Clearly, we do not want the data-triggered threads to create unwanted data races. We allow threads to write global data, as in our example code. 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 store and the main thread join point, it is often easy to verify the absence of data races.

In our current design we do not allow the support thread function to take any argument other than the implicit argument – the triggering address. This greatly simplifies the hardware (no argument storage in the thread queue, no need to verify that the skippable-region arguments match the DTT arguments) and did not impact our ability to transform the code regions we targeted. The support thread can still share data with the main thread through memory.

5.2. 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 our current model, a trigger can appear in two places, in a structure declaration or a variable declaration. After identifying the data structures incurring redundant memory operations, the programmer can attach the pragma `#trigger` to the definition of these data structures or variables. The format of the pragma is:

```
#trigger function_name()
```

In a variable declaration, we are specifying that the thread be initiated any time that variable is changed (Figure 4(a)). In this way, we can attach a thread to specific

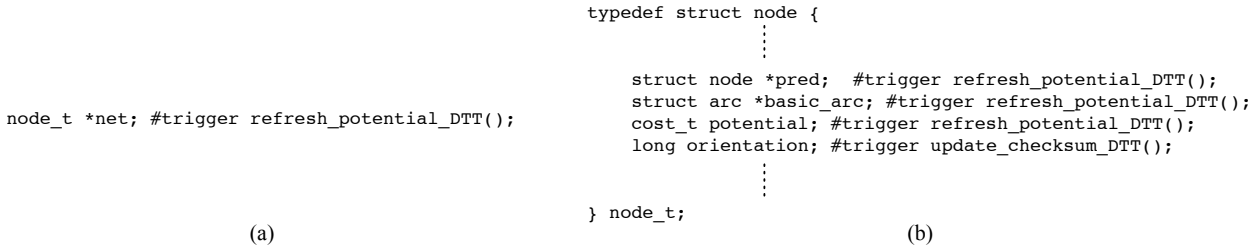

Figure 4. Two possible ways to specify triggers. (a) uses a trigger attached to a specific variable, (b) uses a trigger attached to a particular field of a defined data structure.

variables of a type. On the other hand, when we declare a trigger in a structure declaration, we can attach the thread to a particular element of the structure (Figure 4(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. In our `refresh_potential` example, we use the second mechanism because only modifications to `potential` and pointers to linked nodes (e.g., `basic_arc` and `pred`) will change the value of `potential`. Whenever one of these fields changes, we will initiate the execution of the `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.

The compiler will replace all store instructions that may touch these data structures with a *tstore* instruction, described in the next section. 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.

5.3. Composing data-triggered threads

Each data-triggered thread is written as a function in a conventional programming language. The data-triggered threads can accept the triggering address as an input and share data with the main thread. The data-triggered thread can also have a return statement. The return register value, if there is one, will be kept in the *thread status table* (we will explain the thread status table in detail later) and sent to the main thread only when the main thread reaches that point in the program where it will use the computation of the triggered thread.

In our current instantiation of the data-triggered thread programming model, (often identical) code appears in two places: in the definition of the data-triggered thread, 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 data-triggered thread function, the programmer uses the pragma

`#DTT block_name`

to specify the code blocks that can be skipped after the data-triggered thread is executed.

In our proposed model, we allow more than one data-triggered thread 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.

In our `refresh_potential` function example, the statements in the while loop exhibit a significant quan-

```
// The data structure
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;

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

```
// The data triggered thread
#DTT refreshPotential
long 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 ) {
            tmp = node->sibling;
            if(tmp) {
                node = tmp;
                break;
            }
            else
                node = node->pred;
        }
    }
}
```

Figure 5. An excerpt of modified `refresh_potential` function

tity of redundant loads. As shown in Figure 5, this piece of code can be replaced by the data-triggered thread `refresh_potential_DTT`. The data-triggered thread performs the update of the `potential` field of a modified node and its succeeding nodes so that the program does not need to perform the `refresh_potential` function again if the value of the `potential` field in any node is not going to change further. We could also try to apply a software technique like memoization [6, 21] 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.

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 the 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

`#cancel`

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

Data-triggered threads occupy the same address space as the main thread, and can read and write anything in memory. The support thread has its own stack so that it can maintain its own local variables and even make function calls.

For this paper, 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 1 lists the number of static instructions of our DTTs. The average length of our DTTs is 145 instructions.

We also list our modifications to the C SPEC 2000 benchmarks in Table 1. 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.

6. Architectural support

The data-triggered thread 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 our data-triggered thread execution model, we propose some architectural changes to the baseline processor. These include the *thread status table*, the *thread queue*, and the *thread registry*. In addition, a set of new instructions, *tstore*, *tspawn*, *tcancel*, and *treturn* are added to the existing instruction set architecture. In this

Benchmark	Data triggers	Avg. static DTT inst.	Computation performed by data-triggered threads
ammp	last, naybor	193	Some code from a_number function to recompute total number of nodes and code from mm_fv_update_nonbon function to refresh atomall
art	f1_layer[].P	313	Some code from train_match function to refresh f1_layer[].Y
bzip2	ss	39	Computes the value of bbStart, bbSize and shifts
crafty	search	101	Some code from Evaluate function to precompute the new score
eon	a_MR and a_VHR in mrSurfaceList::viewingHit method	67	The constructor of the ggMaterialRecord or mrViewingHitRecord class
equake	time, disp[]	57	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.
gcc	reg_rtx_no	4	The computation of max_reg_num function
gzip	strstart, hash_head	30	The computation of longest_match function
mcf	node_t	35	Some code from refresh_potential function to update the subtree leading by the touched node
mesa	i0, w0, and w1	203	Generating new R, G, B, and alpha values
parser	randtable and inputs of count function	55	The computation of hash function
perlbmk	PL_op	46	Pre-executing the function specified by PL_op
twolf	new_total of dimptr	159	The computation of new_dbox_a function
vortex	EmpTkn010 and PersonTkn	168	The computation of PersonObjs_FindIn function
vpr	heap	30	The computation of my_allocate function

Table 1. Modifications to benchmarks

section, we will introduce these new architectural components and discuss how they enable the data-triggered thread execution model.

6.1. ISA support

For this implementation of data-triggered threads, we assume four new instructions added to the instruction set architecture. The primary addition is the *tstore* instruction. It causes a thread to be generated if the store modifies memory. We also examined 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 refresh_potential example, we can trigger on a change to the pred or 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 *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 *tstore* instruction, hardware detects whether the store is silent or not. A good description of the hardware to detect silent stores is in [18]. Note that we do not simulate the performance gains from short-circuiting silent stores except for *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 writing to the *thread queue*.

The code executed in the data-triggered thread has some implicit arguments (which become live-ins) and a result. Because the data-triggered thread is specified as a function, the live-ins are 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.

We must be able to associate a *tstore* with the correct data-triggered thread, and a store instruction typically has no unused fields. We assume a hardware structure called the *thread registry*. An entry in the registry contains the PC of the data-triggered thread, the *start PC* of the skippable code in the main thread, the *destination PC* which denotes the end of the skippable region and the new PC after the region is skipped. It is assumed that this table is filled at the beginning of execution by writing to special hardware registers. We never use more than three entries in the thread registry. To associate a *tstore* instruction with a data-triggered thread, we must follow it with a *tspawn* instruction, whose only argument is an index into the thread registry. Data from the thread registry will be used to fill the *thread queue* and *thread status table*, to be described in the next section.

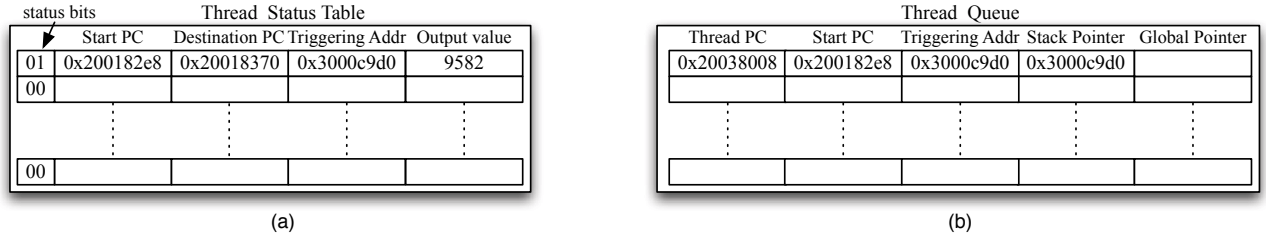

Figure 6. The design of thread status table (a) and thread queue(b)

We assume the hardware injects move instructions right after the *tstore* to transfer the implicit live-ins (sp, gp) from registers to the Thread Queue. We currently have two kinds of *tstore* instructions: for the declaration in Figure 4(a), we pass the effective address to the TQ. For the structure-based declaration in Figure 4(b), we pass the base address of the structure to the TQ. In this case, we just need to ensure that the compiler constructs 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 *tstore* does not alter memory (not actually known until the instruction commits), the *tspawn* instruction is ignored and the transfers do not take place. Because our results 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 encounters a *tcancel* instruction, the running support thread will terminate its execution immediately. As discussed previously, this enables us to create a data-triggered thread 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 executes a *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. 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 thread code. This would allow its live-out to be written into the TST to bypass future redundant computation.

6.2. New hardware structures

To efficiently support our data-triggered thread model, we add the *thread queue* (TQ) and the *thread status table* (TST) as shown in Figure 6, in addition to the *thread registry* (TR) already described. The TST is the same size as the TR (the TR is static, entries in the TST change dynamically), and the size of the TQ varies dynamically with the number of in-flight DTTs.

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 registry – 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 data-triggered thread 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 neccessary 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 the data-triggered thread is still *spawning*, we will remove the

data-triggered thread from the TQ and execute the code in place, and the data-triggered thread 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 [17, 18] which incurs no extra delays.

7. Methodology

In this paper, we evaluate our data-triggered thread architecture using a modified version of SMTSIM [28]. 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 data-triggered thread 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) [29]. We assume the CMP platform is a dual-core processor in which each core has a private instruction cache and data cache, 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 [24] 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 [24]. All simulation results use the reference input sets. For each benchmark, we rewrote the functions containing the most redundant load instructions using the proposed data-triggered thread model as descibed in Section 5.3.

8. Results

Figure 7 shows the experimental results of our modified codes running on the data-triggered thread 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%.

The data-triggered thread model running on this architecture achieves a speedup of 5.89 on mcf. As discussed in our running example, 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 data-triggered thread architecture achieves a 40% performance improvement on the SMT platform.

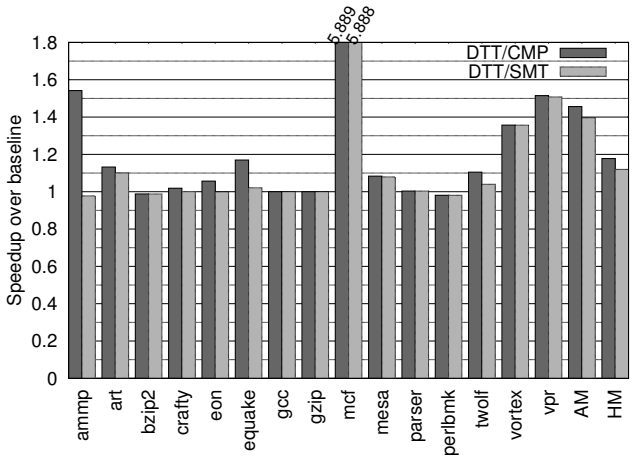

Figure 7. The relative performance of data-triggered threads for the CMP and SMT configurations, relative to our baseline. (HM means harmonic mean, and AM means arithmetic mean)

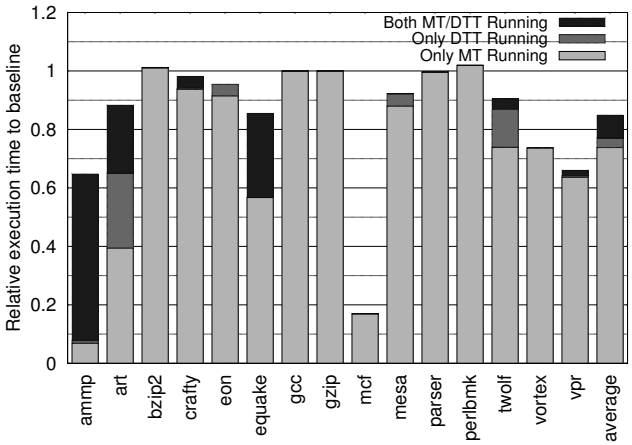

Figure 8. The execution time breakdown for our benchmarks for the CMP configuration.

We further break down the execution statistics in Figure 8. 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 of redundant computation. The experimental results of the two architectures are nearly identical for benchmarks like mcf, mesa, vortex, and vpr. Table 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 data-triggered thread 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 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. 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 targetting 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 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.

It must be pointed out that because our largest per-

	Relative Num. of Dyn. Insts.				Cache Miss Rates					
	DTT/CMP		DTT/SMT		L1			L2		
Name	Main Thread	DTT	Main Thread	DTT	baseline	DTT/CMP	DTT/SMT	baseline	DTT/CMP	DTT/SMT
ammp	0.65	0.51	0.65	0.32	3.26%	3.02%	3.84%	32.31%	22.48%	36.77%
art	0.68	0.34	0.68	0.34	31.09%	20.42%	31.56%	92.01%	88.65%	87.71%
bzip2	1.01	0.00	1.01	0.00	1.23%	1.14%	1.15%	48.21%	48.03%	48.03%
crafty	0.96	0.09	0.96	0.07	1.07%	1.09%	1.24%	3.79%	3.84%	3.50%
eon	0.89	0.05	0.89	0.05	0.14%	0.14%	0.14%	1.84%	2.52%	2.52%
equake	0.60	0.30	0.60	0.30	9.32%	9.34%	11.88%	85.51%	57.62%	53.25%
gcc	1.00	0.00	1.00	0.00	0.78%	0.78%	0.78%	25.29%	25.34%	25.34%
gzip	1.00	0.00	1.00	0.00	4.61%	1.80%	1.80%	0.52%	0.54%	0.54%
mcf	0.56	0.00	0.56	0.00	36.92%	31.40%	31.32%	87.74%	76.82%	76.95%
mesa	0.87	0.01	0.87	0.01	0.34%	0.35%	0.37%	50.34%	19.42%	18.87%
parser	1.00	0.00	1.00	0.00	1.90%	1.86%	1.86%	41.51%	42.48%	42.48%
perlbmk	1.00	0.00	1.00	0.00	0.30%	0.30%	0.30%	6.98%	6.92%	6.92%
twolf	0.89	0.12	0.91	0.12	3.49%	3.50%	3.40%	50.31%	47.26%	53.26%
vortex	0.88	0.00	0.88	0.00	1.20%	0.99%	0.99%	16.66%	10.17%	10.19%
vpr	0.86	0.02	0.86	0.02	3.30%	2.96%	2.99%	51.23%	53.30%	52.86%
average	0.86	0.10	0.86	0.08	6.60%	5.27%	6.24%	39.62%	33.69%	34.61%

Table 2. Relative number of dynamic instructions and cache miss rates for DTT configurations

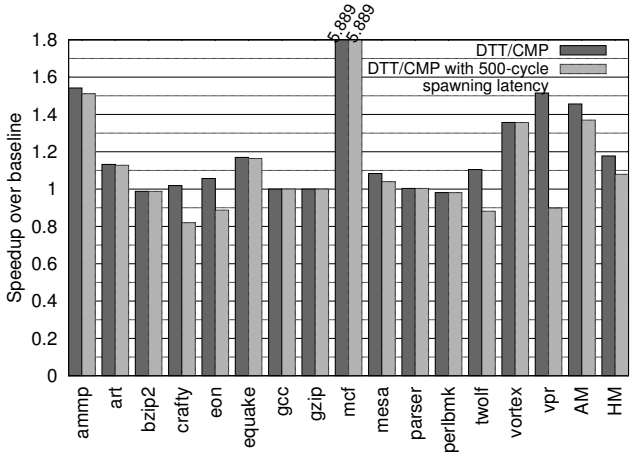

Figure 9. The performance with thread spawn latencies of 10 vs 500 cycles.

formance gains were due to reduced computation and an even greater reduction in total cache misses (power-hungry DRAM accesses), the energy gains resulting from the data-triggered thread architecture will be quite high – larger than the performance gains, in general.

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 9 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 no slack between the data-triggered thread 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.

9. Conclusions

This paper presents the *data-triggered thread* execution and programming model. With data-triggered threads, 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. By specifying computation in a data-triggered thread, the computation is only performed when the data gets changed, eliminating redundant executions. We show that 78% of the loads in the C SPEC benchmarks are redundant and create unnecessary computation. By making small changes to existing C programs, we achieve speedups with data-triggered threads as high as 5.89, and averaging 1.46.

Acknowledgments

Several of the key ideas in this paper had their origin in early conversations with Jamison Collins of Intel. The authors would also like to thank the anonymous reviewers for their helpful comments. This work was funded in part by NSF grants, including CCF-1018356, and support from Intel Corporation.

References

[1] S. Amamiya, M. Izumi, T. Matsuzaki, R. Hasegawa, and M. Amamiya. Fuce: the continuation-based multithreading processor. In *Proceedings of the 4th international conference on Computing frontiers*, pages 213–224, May 2007.

[2] B. S. Ang and D. Chiou. StarT the Next Generation: Integrating global caches and dataflow architecture. In *CSG Memo 354, Computation Structures Group, MIT Lab. for Comp. Sci*, 1994.

[3] Arvind and D. E. Culler. Dataflow architectures. *Annual review of computer science vol. 1, 1986*, pages 225–253, 1986.

[4] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *33rd Annual International Symposium on Computer Architecture*, volume 0, pages 302–313, June 2006.

[5] R. Bodík, R. Gupta, and M. L. Soffa. Load-reuse analysis: design and evaluation. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 64–76, New York, NY, USA, 1999. ACM.

[6] 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.

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *34th International Symposium on Microarchitecture*, pages 306–317, December 2001.

[8] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[9] 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.

[10] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *2th Annual International Symposium on Computer Architecture*, pages 126–132, 1975.

[11] P. Evripidou. D3-Machine: A decoupled data-driven multithreaded architecture with variable resolution support. *Parallel Computing*, 27(9):1197 – 1225, 2001.

[12] J. Huang and D. Lilja. Exploiting basic block value locality with block reuse. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 106–114, January 1999.

[13] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A design study of the EARTH multiprocessor. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pages 59–68, June 1995.

[14] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *15th Annual International Symposium on Computer Architecture*, pages 131–140, May 1988.

[15] 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.

[16] C. Kyriacou. Data-Driven Multithreading using conventional microprocessors. *IEEE Transaction on Parallel Distributed System*, 17(10):1176–1188, 2006.

[17] 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.

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

[19] M. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *SIGOPS Operating Systems Review*, 30(5):138–147, 1996.

[20] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 77–84, New York, NY, USA, 1998. ACM.

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

[22] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

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

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Comference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[25] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[26] G. S. Sohi and A. Sodani. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[27] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *27th Annual International Symposium on Computer Architecture*, pages 1–12, March 2000.

[28] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of 22nd Annual Computer Measurement Group Conference*, December 1996.

[29] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, Jun 1995.

[30] W. Zhang, B. Calder, and D. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th international conference on Parallel Architectures and Compilation Techniques*, pages 87–98, September 2005.

[31] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.