In proceedings of the 10th International Symposium on High Performance Computer Architecture, February 2004.

Creating Converged Trace Schedules Using String Matching

Satish Narayanasamy[†] Yuanfang Hu[†] Suleyman Sair[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Department of Electrical and Computer Engineering, NC State University

Abstract

This paper focuses on generating efficient software pipelined schedules for in-order machines, which we call Converged Trace Schedules*. For a candidate loop, we form a string of trace block identifiers by hashing together addresses of aggressively scheduled instructions from multiple iterations of a loop. In this process, the loop is unrolled and scheduled until we identify a repeating pattern in the string. Instructions corresponding to this repeating pattern form the kernel for our software pipelined schedule. We evaluate this approach to create aggressive schedules by using it in dynamic hardware and software optimization systems for an in-order architecture.*

1 Introduction

In-order and VLIW architectures rely upon the compiler to create efficient schedules so as to attain faster clock cycles and power efficiency. Software pipelining [17] was proposed as a means to expose more ILP to these architectures. Even though there have been numerous approaches [2, 3, 10, 12, 21, 4] to software pipelining, it has remained a complex problem.

Kernel Recognition [2, 3, 4] algorithms are one class of solutions to do software pipelining. These algorithms simultaneously unroll and schedule the loops. Scheduling is done by assuming a resource model for the target machine. The process of unrolling and scheduling is carried out until a repeating pattern, that is a kernel, is detected in the resulting schedules. Once a kernel is detected in the unrolled and scheduled loop, a backward branch to the beginning of the kernel is added to the end of the schedule. This schedule becomes the new loop with the kernel as the loop body. Dynamically forcing a repeating pattern to materialize without compromising the efficiency and code size of the schedules, and detecting such patterns has been important issues with this class of algorithms.

In this paper, we examine using string matching to help guide kernel recognition when constructing software pipelined schedules for loops. We create a schedule for a candidate loop, by simulating the effect of executing the loop out-of-order. This results in an unrolled and scheduled loop, where scheduling is done across multiple iterations. The sequence of instructions in this schedule are broken into *Trace Blocks*, where each trace block is the sequence of instructions between two backwards taken loop branches. The trace blocks are then hashed down to string IDs, and in this way the whole schedule is converted into a string. We can then use a string matching algorithm for kernel recognition to find a repeating pattern in this string. We call these *Converged Trace Schedules*, because we find that the string patterns seen in the scheduled trace converge to repetitive patterns after several scheduled iterations.

Our algorithm can be implemented offline in a profile guided compiler or can be implemented in a dynamic optimization system. In this paper, we evaluate dynamic implementations for Converged Trace Schedules. We evaluate (1) using a software thread to generate the converged schedule by sharing (context switching on) the same processor with the program's main thread being optimized, (2) using a software thread running on a different processor (in a multi-processor) in parallel with the main program in order to generate the converged trace schedules, and (3) using a special purpose co-processor to generate the converged trace schedules. The performance of these approaches are compared with previously proposed aggressive dynamic trace scheduling techniques [18].

The rest of the paper is organized as follows. In the next section we discuss related work. Section 3 gives an overview of our architecture. Section 4 details the converged trace formation algorithm, and three possible implementations of the algorithm are described in Section 5. Section 6 elaborates on the additional hardware features needed. In Sections 7 and 8 we discuss our methodology and present the results. Finally we conclude in Section 9.

2 Related Work

In this section, we discuss several relevant research efforts to do software pipelining. We also present prior work related to trace-based dynamic optimization systems and micro-architectural features needed to implement them.

2.1 Software Pipelining

Various algorithms for software pipelining exist [3]. Broadly these algorithms can be classified under the following three categories: Enhanced Pipeline Scheduling [10, 12], Modulo Scheduling [21, 22] and Kernel Recognition [2, 4]. A detailed survey on these algorithms can be found in [3].

2.1.1 Enhanced Pipeline Scheduling

For Enhanced Pipeline Scheduling techniques, instructions are moved forward in the execution schedule. As these instructions are moved out of the loop body into the *prolog* portion of the schedule, they are also moved back into the bottom of the loop body as operations from a later iteration. Even though this class of algorithms prevent explosion of code size and can be efficient, the algorithm is quite complex.

2.1.2 Modulo Scheduling

Modulo scheduling uses a different approach in that instead of free code motion like in the previous approach, it first assumes an *Initiation Interval* (II). II specifies a constraint that when the desired schedule is executed repeatedly at intervals of II, there should not be any stalls due to resource conflicts. Under this constraint, the algorithm schedules each instruction in one iteration and tries to arrive at a legal schedule for that iteration. This legal schedule is the kernel that can be iterated indefinitely without violating any dependency constraints. If no legal schedule can be formed for the assumed II, then modulo scheduling is tried for greater values of II. Usually, loops are unrolled before the application of modulo scheduling to enable arriving at legal schedules for lower IIs. While this algorithm is elegant, determining minimum II has been an issue and one approach([22]) has been to start with the minimum possible II (determined by the critical path in the loop) and iteratively increment II until a legal schedule can be formed or an upper bound is reached.

2.1.3 Kernel Recognition

Kernel Recognition algorithms [2, 4] simultaneously unroll and schedule the loop. This process is continued until a repeating pattern is detected in the schedule. These kernel recognition algorithms are capable of forming very efficient schedules but can result in schedules with large code sizes as a result of unrolling.

Recognizing when a pattern has formed and aiding efficient formation of such a pattern is essential for these algorithms. To recognize repeating patterns, the state of the schedule at every previously scheduled instruction is maintained. The state of the schedule at a specific instruction represents the information specifying what resources are available to schedule instructions in succeeding instruction slots. When the current state matches one of the previously encountered states, this indicates that there might be a repeating pattern in the schedule. Maintaining and comparing these states has been a complexity issue. In [4], the authors simplify the state comparison problem by hashing the states and comparing the current state with only those previous states with an identical hash value. But still states corresponding to every hash value need to be maintained in the proposed techniques.

Our algorithm takes a different approach in constructing the schedules and finding repeating schedules. To detect repeating schedules, instead of maintaining and comparing with the previous states of the scheduler while scheduling each instruction, we create a string of identifiers. In addition, each identifier representing the scheduled instruction sequence between two backward branches, and not the complete state of the machine. If the same schedule (string pattern) occurs over and over again, then the schedule has converged given the resource constraints and latencies and we can use this to find the kernel.

2.2 Hardware-Based Trace Scheduling

The trace cache [5, 14, 15, 20, 23] stores traces of frequently executed sequences of instructions into physically contiguous memory locations. These traces can be formed in a fill unit which receives the sequence of committed instructions. While forming these traces in the fill unit, dynamic regrouping of instructions can be performed. Optimizations that can be applied include re-association, constant propagation, instruction collapsing and instruction scheduling which are much less aggressive when compared to the technique we are exploring in this paper.

Franklin and Smotherman [13] also proposed using a fill unit to dynamically compact the stream of retired scalar instructions into VLIW instructions. They would then store the individual VLIW words in a *shadow cache* to be used in the future. The major difference between this technique and ours is that they store single VLIW instructions in the shadow cache, not addressing the fetch bottleneck. Also caching on a single VLIW word basis limits the amount of parallelism that could be extracted when compared to having larger units for caching.

An efficient way to implement various instruction scheduling techniques dynamically is the *instruction path co-processor* (ICOP) concept [9]. ICOP is a programmable on-chip co-processor that operates on the host processor's instruction stream to transform them into a new schedule to enable more efficient execution. It is located off the processor critical path which makes it ideal for implementing dynamic backend optimizers when combined with its flexible nature. The notion of introducing a co-processor to the back end of the pipeline so that it could exploit code reuse while not interfering with the critical path of the host processor is the motivation for the co-processor implementation of our scheduling algorithm, which is one of three implementations we examine.

Nair and Hopkins [18], introduced the Dynamic Instruction Formatting (DIF) cache to perform aggressive code scheduling. In the DIF paradigm there are two execution cores on the chip: the primary engine, which is a simple in-order processor, and the parallel engine which is a fast VLIW core acting as an accelerator executing dynamically scheduled groups of instructions(i.e. VLIWs). The primary engine's main role is to train the parallel engine as described below. Additionally, its secondary function is to act as a fall back mechanism to process unexpected events such as exceptions. As we will explain in subsequent sections, the notion of falling back to the simpler mode of operation to implement precise exceptions or recover from branch mispredictions is used within our adaptive scheduling framework as well. However, we do not rely on a faster execution core to achieve speedups. We have a single in-order core that is fed from one of two different instruction streams (I-Cache or CT-Cache) based on the mode of operation at a given cycle. The speedups we achieve are strictly attributed to the better scheduling of the original instruction order.

Similar to the DIF cache, Black and Shen [7] developed

the Turboscalar architecture which uses the Dynamic Instruction (DI) Cache . This cache is part of a technique to build wide and shallow superscalar machines without paying the price of a longer clock cycle time. The DI Cache is very similar to the DIF cache, however, the underlying execution core is much more aggressive (i.e. out-of-order superscalar core). Turboscalar utilizes a single execution engine, which is fed from one of two different pipelines in a given cycle.

The major difference between DIF and Turboscalar concepts and our technique is the granularity at which the instruction scheduling decisions are made. DIF and Turboscalar assume maximum instruction windows of 48 or 16 entries respectively to schedule instructions from. In contrast, our scheduling technique is more aggressive (i.e. analyzing instructions in the order of hundreds), resulting in converged schedules along the same vein as software pipelined loops. We allow aggressive forward speculation along with scheduling across procedure calls and returns. Aggressive speculation coupled with register windows enables highly parallel schedules to be created. The success of our converged trace scheduler depends on the size of the scheduler's available pool of instructions to be in the 100s to 1000s. In comparison, the largest DIF window of 48 instructions, while performing aggressive speculation, cannot provide an aggressively software pipelined schedule.

2.3 Handling Exceptions with Aggressive Speculation

In [11], Ebcioglu and Altman describe the essential architectural features needed for aggressive speculation. They describe keeping an additional exception tag bit, indicating that the register contains the result of an operation that caused an exception for each register. In this case, an instruction that was scheduled as speculative will not cause an exception. Instead it sets the result register's tag bit, and the exception is propagated in this manner through speculative instructions until the register is possibly consumed by a non-speculative instruction. At that point, the exception occurs. This allows moving loads above branches and stores. They keep a non-speculative version of the load in its original place to process the exception if one occurs.

In [19], Nystrom et al. describe a precise speculation technique that allows dynamic optimization systems to perform aggressive code reordering and speculation while ensuring that exceptions are taken in their proper order. It does this by providing a separate speculative register file to keep the results of speculative instructions. With precise speculation, optimization and code reordering can be applied beyond branch boundaries or excepting instructions.

3 Processor Overview

In this section we give a brief overview of the host in-order processor core we assume. This processor has a dynamic optimizer that implements our *Converged Trace Scheduling* algorithm.

3.1 High level Operation

Figure 1 shows the pipeline organization of our processor model. Initially, instructions are fetched from the instruction cache. Committed instructions from the in-order processor are indexed into a CT-Schedule driver, which keeps track of backward loop branches. The CT-Schedule driver decides when to start and stop sending instructions to the scheduler to generate schedules for candidate loops. The goal is to only schedule a small fraction of the executed instructions, in order to keep the overhead of dynamic schedule generation to a minimum.

In scheduling mode, committed instructions are collected and scheduled by the CT scheduler. A converged trace schedule is formed using our converged trace scheduling algorithm described in Section 4. For dynamic creation of the Converged Trace Schedules, the scheduling algorithm can be implemented either in software running on the same or another processor/context, or in hardware with a co-processor. We provide results for both implementations in Section 8. When creating the converged trace schedule, the scheduler takes into account the architectural parameters used for the target in-order processor such as cache hit latency, memory access latency, processor issue width, number and type of functional units, etc. This enables it to generate schedules optimized for the target architecture.

The newly formed converged trace schedules are stored in a *Converged Trace Cache* (CT-cache) for future use. Each of these schedules are associated with a trigger instruction, which is kept track of in the *Fetch Driver*. When this trigger instruction is the predicted as the next fetch block PC, the in-order processor switches to trace mode and starts consuming instructions from the CT-cache in the scheduled order instead of from I-cache. The fetch driver is also responsible for predicting the next block in the scheduled loop and for providing the next PC every cycle. We remain executing in trace mode until (1) we leave the scheduled code region, (2) a branch misprediction occurs, (3) a memory mis-speculation occurs in the scheduled trace, or (4) an exception is raised. When these occur, we return to in-order fetch until we reach the next trace scheduled region.

3.2 Register Windows

Since the goal of the converged trace schedules created by our scheduler is to significantly increase parallelism, the schedules may need to increase the number of live registers to expose the desired ILP. To accomplish this we use register windows, similar to the approach used in [6, 18] to support aggressive hardware instruction speculation. The execution of a loop branch from the trace cache increments the active register window. Each register window size is equal to the number of logical registers in the ISA, similar to the original in-order core. The number of register windows needed is the maximum number of loop iterations we allow the scheduler to speculate across, plus one for the default in-order execution (called RW-0). The in-order core will use the register windows for the converged trace schedules to expose more ILP. Because of these multiple windows, the architecture needs to use a register map for operands to determine which window to find their values in. Note that we do not have to do register allocation as in an out-of-order processor, since the renaming is provided automatically by the register windows (i.e. there is no free register pool).

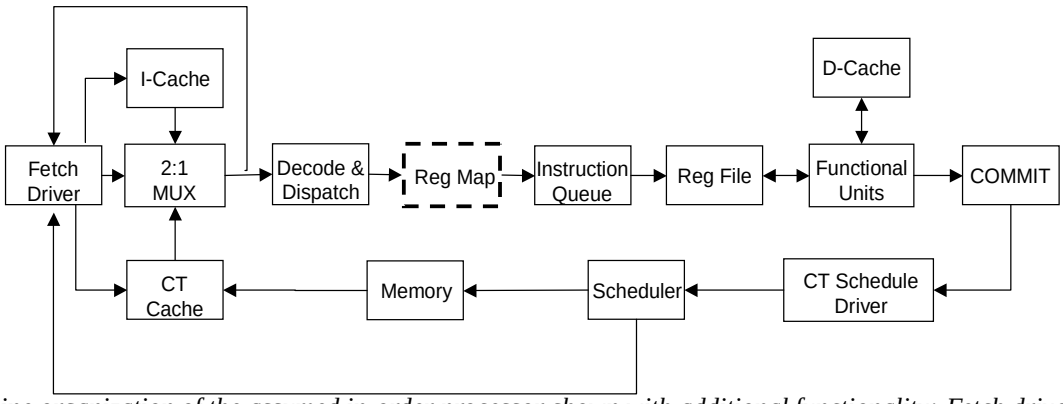

Figure 1: Pipeline organization of the assumed in-order processor shown with additional functionality: Fetch driver, CT-Schedule driver, CT-Cache that are needed to implement converged trace scheduler and enable the processor to use these schedules. The register mapping stage drawn with dashed lines and a larger register file are the two main effects on the in-order core's processor for using the aggressive converged trace schedules.

4 Converged Trace Scheduling Algorithm

In this section we describe our feedback-directed Converged Trace Scheduling algorithm. These are called converged trace schedules (CTS) because as we schedule over multiple iterations of the loop, the resulting schedules tend to naturally converge to a repeating pattern. The instructions corresponding to this repeating pattern constitute the kernel of CTS.

4.1 Algorithm Overview

The software pipelined Converged Trace Schedule (CTS) that we create for a candidate loop contains only a kernel and a prolog. No epilog code is needed because of the hardware trace mispeculation support that we have in our host processor. This is described more in detail in Section 6.4.1.

In this section we describe how we generate the kernel and prolog portions of CTS for a candidate loop. Figure 2 shows the main steps in the scheduler to create converged schedule for a candidate loop. Below is a high level summary of each component of the algorithm, followed by sections that describe each in detail:

- Schedule Window - An in-order trace of instructions resulting from the execution of multiple iterations of the candidate loop is the input to our CTS algorithm. These instructions are annotated with their register and memory dependency information and also with the information on their latency of execution.
- ILP Scheduler - Similar to a scheduler in an out-of-order processor, the ILP Scheduler aggressively schedules instructions out-of-order over the instruction Scheduling Window using a detailed architecture model. While scheduling, the scheduler adheres to additional rules which enables us to generate schedules that converge to a recurring pattern.

  This resulting scheduled trace is broken into what we call Trace Blocks using taken backward branches as delimiters. A *Trace Block* is a sequence of instructions between two adjacent taken backward branches in this scheduled trace.
- Creation of Hashed Trace Patterns - As the trace blocks are formed, they are hashed down to a unique ID. A hash function is used such that every time when a trace block is formed with the exact same sequence of instructions it will get the same ID. This creates a sequence of IDs forming a trace string. This is then passed to the pattern finder to search for recurring patterns.
- Finding Repeating Strings - We used a modified version of the Aho-Corasick [1] algorithm for finding consecutively repeating string sequences. When a pattern of IDs (trace blocks) repeat consecutively, we know that the schedule has converged. The trace blocks corresponding to these repeating string of IDs represent the kernel of a software pipelined loop.
- Forming Prolog for the Converged Trace Kernel - The converged repeating sequence found above will need a Prolog code sequence to start the initial iterations of the loop before starting to repeatedly execute the kernel portion. Therefore, a prolog block of instructions is created for the kernel and is pre-appended to it to get the final trace.
- Converged trace compression - The final step of the scheduling algorithm is to compress the kernel if needed. The kernel may consist of repeating trace block patterns. In this case, a fixed count loop is put around this trace block pattern. This scenario is possible especially when we are generating schedules for nested loops. Consecutively repeating trace block patterns in the kernel, are reduced to one pattern, and the number of times it repeats is recorded with the kernel to be used by the trace cache fetch driver. The fetch driver will be explained in Section 6.3.
- Trace output - For the dynamic implementations we examine in this paper, information to trigger and execute the converged trace is inserted into the fetch driver, and the converged trace is stored in memory on a page reserved for dynamically generated code sequences.

4.2 Schedule Window

The input to our algorithm is an in-order trace of instructions resulting from the execution of multiple iterations of the candidate

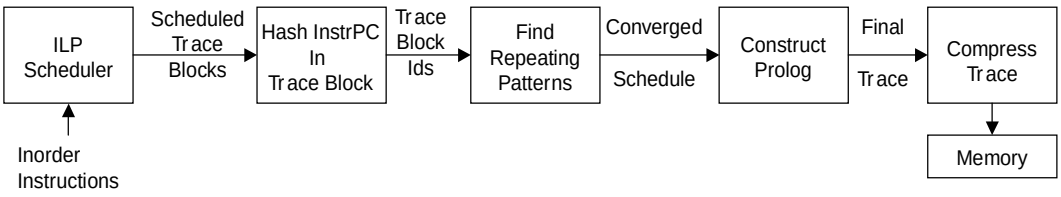

Figure 2: Components of the Converged Trace Scheduling algorithm.

loop. In the case of dynamic optimization systems, this trace is collected from the sequence of committed instructions in the processor. For offline analysis it can be collected through profiling or simulation. In fact, only the trace of branch PCs need to be collected as the sequences of instructions executed can be derived from this.

For a loop, we build up a large window (several hundreds) of instructions to create a schedule from. Instructions are inserted into this *Schedule Window* in-order from traversing the loop, possibly for many iterations. The instructions are annotated in the window with exactly how many cycles it will take to execute the instruction, and with both register and memory dependency information. We do not annotate register write-after-write and write-after-read dependencies. This is not needed because during scheduling, since we assume perfect register renaming which is made possible by using register windows to support the speculative scheduling. This is described more in detail Section 3.2.

4.3 ILP Scheduler

The job of the ILP scheduler is to consume in-order sequences of instructions from the schedule window and produce a trace of scheduled instructions which will then be given as input to the pattern finder. ILP scheduler schedules instructions similar to a scheduler in an out-of-order processor. When an instruction is brought into the scheduling window its dependences are hooked up to its producing instructions. As dependences for instructions get resolved they are scheduled greedily. They are then removed from the schedule window out-of-order as they schedule, and new instructions are brought into the scheduling window.

Following are the key features of our ILP scheduler that aids in forming efficient schedules that would eventually converge to a repeating pattern.

- A large schedule window of size 500 is used for choosing instructions to schedule out-of-order. This is large enough to hold a single instance of the largest loop body we observed for the programs we studied (see Table 1).
- All branches are kept in original program order. A branch is not allowed to be speculated and scheduled above a prior branch.
- Speculation is controlled in order to generate a schedule that converges faster to a repeating pattern and is smaller in terms of the number or loop traversals it represents. This is achieved by limiting the number of branches, say X, that any instruction can be speculated above. An instruction can be scheduled only if the Xth preceding branch has already been scheduled. Note that this also implies that all of the branches before the Xth prior branch have also been scheduled, since branches are scheduled in original program order. One has to carefully choose X, as a smaller value would mean that the schedules are less aggressively speculated, which in turn limits the scope for optimization. In the results in this study, we set X to six.
- Aggressive memory speculation is allowed moving loads above stores that were found to have independent addresses in the trace of instructions given as input. We assume hardware support, described in Section 6.4.2, to detect speculation violation during execution and recover from it.
- It is necessary to eliminate non-determinism to generate converged schedules. Hence a latency for each load is individually chosen and that same latency is used for scheduling throughout the loop's scheduling. The loads are assumed to finish their execution when that latency is up no matter whether the load hits or misses in the cache in the original in-order execution. This latency determines how far away the first use of the load will be scheduled from the load.
- Reproducibility is a must for obtaining converged trace schedules. Hence these schedules are generated by enforcing the following important restrictions: (1) if a given set of instructions have all of their dependencies met, they are issued in instruction fetch order, (2) all load instructions are assigned a fixed execution latency, and (3) speculation above conditional branches and function calls/returns are limited to a certain range, and (4) we do not start scheduling until the scheduling window is full.
- We assume rotating register windows, as described in the prior section, so during scheduling we assume perfect register renaming.

4.3.1 Load Scheduling

The ILP Scheduler schedules a load either as a hit or a miss with a fixed latency. This is enforced by fixing the execution time for a given load instruction throughout the whole process of trying to generate a converged trace. This means that the load is assumed to take N cycles in the scheduler to execute no matter what the real latency is during a scheduling period.

Two straightforward choices to examine for a load is to schedule it as a hit, where N equals the hit latency, or as a miss all the way to main memory where N is equal to the main memory latency. The latter case, allows as many independent instructions as possible to get between the load and its first use (pending the limits on speculation described above), before that first use is scheduled. This can be advantageous because a stalled first use in an in-order processor will stall the pipeline until the load comes back from the memory hierarchy. A drawback of delaying

the instructions that depend on the load is potentially impeding with the critical path. Since loads are usually on the critical path, instructions relying on them are also part of the critical path.

For the results in this paper, to decide on whether to schedule a load as a hit or miss, we use a variation on a common heuristic used by static schedulers in existing compilers, but apply it dynamically. Our scheduler classifies a load as critical if P% of the instructions in the fetched window after the load are directly or indirectly dependent upon the load. This identifies loads that can clog the schedule and limit instruction level parallelism if their uses are not scheduled as a load hit. We examined a range of values for this heuristic. We found that classifying a load as schedule critical, when at least 20% of the 100 instructions fetched after the load are directly or indirectly dependent upon it, performed well for all of the programs examined. If a load has this property, then the load's uses are scheduled as if the load hit in the L1 cache. If the load did not satisfy this criteria, then its uses are scheduled as if the load misses all the way to main memory. We examined many other heuristics, from using dynamic sampling of the actual load latency to other static heuristics, and found that the above static heuristic consistently performed the best.

This heuristic is similar to one proposed by Kerns and Eggers [16], where they scheduled instructions based on an estimated amount of load level parallelism in the program. They calculated load level parallelism as the number of instructions that may execute in parallel with each load instruction.

4.3.2 Register Windows and Assigning Register Window Offsets

Since we scheduled the trace of static instructions only obeying true read-after-write dependencies, we need to deal with the conflicts due to false register dependences. To deal with this, we assume register windows in the target architecture to provide an efficient form of register renaming/allocation for our converged schedules.

For each instruction, we keep track of the number of loop branches the instruction was speculated above. This may be from 0 to the maximum branch speculation depth. The offset (number of speculated loop branches) is stored with each instruction as it is scheduled by the ILP scheduler. This offset will be used to indicate which *future* register window to store the register value in, since the speculative instruction is coming from a future loop iteration.

When a register definition from the trace cache is executed, it assigns its definition to the register window defined by adding the current instruction window number to this future register window offset. This effectively stores the register definition in the register window corresponding to the loop iteration where the instruction was speculated from. The idea here is for speculated instructions to store their values in the register window belonging to the loop iteration from which they came.

When processing instructions in-order from the instruction cache (referred as non-trace mode), the processor performs all of its writes to a default register window called register window zero (RW-0). For an instruction to determine where in the register file to obtain its operand input values, it must read a register map (see Section 3.2), to determine if the register definition comes two possible locations. From either RW-0 (default in-order window), or the current instructions future register window. If the register definition is to come from a register window other than RW-0, the register window offset of the current instruction is added to the current/active register window to determine what future register window to read the value from. In non-trace mode, the register map will point to the exact window to find the register in (RW-0 or the last valid register window used by the converged trace schedule). This ensures correct register values are read when we have terminated fetching from the converged trace cache. To update the map correctly for this, the loop exiting branch has a mask associated with it in the trace cache indicating what the live out registers are. The register map is then correctly updated when we pay the branch misprediction penalty for kicking out of the trace cache.

4.4 Creating Trace Hashed Patterns

As the scheduled instruction sequences are produced, the sequences are divided into trace blocks. A backward branch marks the end of a trace block, and the very next scheduled instruction marks the start of the next trace block. As the scheduler produces each trace block, we hash the PCs corresponding to the instructions in the trace block to create an ID to represent that trace block. A hash function is used, so that the same sequence of instructions in a trace block will produce the same ID. The hash function we use consists of a set of shift and xor operations accumulating over every instruction in the order they appear in the trace block.

The string of trace block IDs are then fed to the next stage to find converged patterns. For every unique ID, its corresponding trace block needs to be stored in a cache. Later when a repeating pattern of IDs is found, trace blocks corresponding to that pattern are retrieved from this cache and the kernel is constructed. As mentioned earlier, a trace block can be retrieved by storing just the sequence of branches in the block.

4.5 Finding Converged Trace Block Patterns

Our algorithm is derived from the string matching algorithm of Aho-Corasick [1]. The aim is to find a sequence of trace blocks that repeat consecutively. We create a tree for each unique trace block ID encountered, with this ID as the root. Each tree keeps track of patterns that start with its root ID and traversing it from its root to one of its leaves gives us a pattern that had occurred in the past. Also the frequency of the patterns are tracked. On every input of trace block ID, all the trees that are in the working set are updated to keep track of patterns. A tree will be in the working set if the previous ID encountered is part of some pattern represented by the tree. Trees might contain loops to efficiently capture repeating sub-patterns. The depth of the tree is not allowed to exceed fifteen, as it is highly unlikely for there to be repeating patterns with more than fifteen unique trace blocks. At the end of the scheduling interval, all the trees are parsed to find the patterns that have occurred most frequently. If we manage to find a consecutively repeating pattern, then the resulting

pattern is the kernel. We call the resulting schedule a Converged Trace Schedule (CTS). Later we'll see that such highly efficient schedules are possible for programs with regular loops.

When we don't manage to find a consecutively repeating pattern, we choose the pattern that occurs most often in the collected trace. We refer to such schedules as Pattern Trace Schedules (PTS). For programs with irregular control flow, this type of schedules are more frequently observed.

4.6 Converged Schedule Prolog Formation

For each converged trace segment, we create a trace prolog if needed. A prolog is needed if the converged schedule is a software pipelined loop consisting of instructions that were brought into the kernel from multiple iterations away. Let us say we have an instance of an instruction in the kernel which occurs N iterations later in the original in-order trace. For such an instruction, we need to make sure that its instances belonging to $N-1$ prior iterations are in the prolog in order to start using the kernel without violating dependency constraints. Thus this value $N-1$ for an instruction specifies the number of instances required for that instruction before entering the kernel.

If the kernel contains multiple iterations of the loop, then there will be multiple instances of each static instruction in the kernel. For example, if the kernel contains three iterations it will have three instances of the same static instruction. Let us say, these three instances in the kernel originally belongs to N, $N+1$, $N+2$ iterations (it can be noted that these have to be consecutive iterations) ahead in the in-order trace. Here too, we just need to make sure that we have $N-1$ instances of the instruction in the prolog.

The following are the steps we used to build the prolog:

- We first calculate the value N for each static instruction in the converged trace schedule (kernel). This is easily calculated, since we know for each instruction how many loop branches it was speculated above from the register window offset. In the case when the kernel contains multiple loop iterations, there will be multiple instances in the kernel for a static instruction. The value N for the static instruction will actually be the value N of the first instance of the static instruction in the kernel.
- We next calculate value L as the maximum of the $N-1$ values for all of the instructions in the kernel.
- The prolog is then simply formed by walking over the instructions in the order in which instructions were scheduled in the converged trace schedule. This is done L times, decrementing L at the start of each kernel iteration. At each step, if the $N-1$ count for an instruction is greater than L, then an instance of the instruction is added to the prolog. When L equals zero we will have finished construction our prolog for the kernel.

The same method used above is also used to create a prolog block for a pattern trace schedule.

5 Converged Trace Scheduler Implementations

In this paper, we evaluate the efficiency of our converged trace scheduling algorithm described in the previous section by implementing it as part of a dynamic optimization system. As mentioned in Section 1, we have 3 implementations for such a dynamic optimizer that implements our scheduling algorithm, each at different points on the cost vs. performance curve. These are:

- Serial Software Implementation - Here the scheduling algorithm is implemented in a separate compilation thread that shares the same processor/context as the executing program. Therefore, the scheduler will compete directly for hardware execution resources, and it is vital that we come up with the smallest/fastest schedule possible for each converged loop.
- Parallel Software Implementation - For this implementation we model the compilation thread running on a separate processor on a multi-processor chip. Therefore, the delay for creating a schedule is the time it takes to communicate the trace (in compressed branch history form) to the compilation thread, and generate the schedule.
- Co-processor Implementation - In this implementation we assume a dedicated co-processor along the lines of ICOP [9] is used to generate the schedule. We developed a pipelined implementation of the algorithm, which results in a reduced scheduling delay when compared to the parallel software approach.

In each of these cases, the processor is not allowed to use the converged trace schedule traces until they are fully generated and stored in memory. We evaluate the performance of these various implementations in Section 8.

6 Hardware Support for Converged Trace Formation and Execution

Section 3 gave an overview of the architecture to create and use Converged Trace Schedules. In this section we describe the hardware support needed to execute the converged traces in more detail. The components of our architecture include:

- Converged Trace (CT) Schedule Driver - Determines candidate loops for which schedules need to be formed and thus determines when to start and stop converged trace scheduling algorithm.
- CT-Cache - The CT-cache is a first level cache that stores the converged trace schedules.
- Fetch Driver - Decides when to enter trace mode and start fetching from the CT-cache. When in the trace mode, the in-order processor consumes instructions from the CT-cache instead of the I-cache. The fetch driver also provides nextPC and next block information to drive fetch, and helps the memory system to prefetch instructions into CT-cache.

6.1 CT-Schedule Driver

CT-Schedule driver is responsible for determining candidate loops for which converged trace schedules need to be created. It also determines when to start and stop collecting the trace of instructions that are needed to schedule the candidate loop.

We try to generate converged trace schedules only for loops that are executed frequently. Each loop can be identified by the backward branch that guards it, which we call a Loop-Branch. In the CT-Schedule driver, for every Loop-Branch we keep track of its frequency count, which is the number of times it is executed. In addition, we keep track of the maximum number of times it was taken in a row, which is stored as its trip count. If the frequency and the trip count values for a Loop-Branch are above certain thresholds, then the corresponding loop is a candidate for which a converged trace schedule can be built. For the results in this paper, we use a frequency threshold of 50 and a trip count threshold of 5.

We make a transition into the scheduling mode upon encountering a backward branch that has become a candidate loop branch. This starts the scheduler algorithm described in Section 4 and the trace of committed instructions is given as its input. We switch back to non-scheduling mode either (1) when the loop branch is not-taken, or (2) when it has iterated for more than a threshold time, which was set to 60 for our simulations, or (3) when we have found a converged schedule. Once we have tried to form a converged schedule for a loop, we set the "CT-Tried" bit to ensure that we no longer attempt to generate converged schedules for that loop in future.

6.2 Memory System

Once a converged trace schedule is created, the schedule is memory backed to a set of virtual pages dedicated for holding these schedules. The CT-cache acts as a first level cache from which instructions are consumed by the in-order processor in trace mode. We model a cache of size 8KB, which was large enough to hold our largest trace. As execution transitions between different phases of program execution resulting in the execution of different loops, there may be misses in CT-cache when we start to use a particular trace. To eliminate such misses, we use a next-line prefetcher for the CT-cache. Prefetching is very accurate, since the code for the converged trace schedule is sequentially fetched and executed until there is a backwards branch to re-execute the converged schedule.

6.3 Fetch Driver

The Fetch Driver maintains a table of trigger instructions that is updated when the schedules are formed. In the non-trace mode, the fetch driver monitors the instructions going into the processor from the I-cache. If an instruction is found in the trigger table, then we switch to trace mode. While in trace mode, the in-order processor consumes instructions from the CT-cache. The fetch driver also predicts the next fetch address in the CT-cache. This can be easily calculated by adding a constant stride to the current fetch address except when we are at the end of a trace block. At that point, one has to determine the next trace block to be executed. Since we compressed consecutively repeating sub-patterns in a trace into one code sequence, we need to know how many times that trace block pattern needs to be executed. To solve this, when the converged trace is created, a next block predictor entry shown in Figure 3 is allocated. The tag is the PC of the branch ending the repeating sub-pattern, and the target fetch address is also stored. In addition, we record the frequency ($numberiterations$) of the sub-pattern we compressed with the entry. Each time there is a hit in the next block predictor, we predict the target address until the current iteration count equals the number of iterations. If there is a hit, and these two counters are equal, then the fall through address is predicted and the current count is reset to zero.

The last branch in the kernel is also stored in the next block predictor, with a target address pointing to the start of the kernel. The main difference is that the number of iterations for this branch is set to infinity. So if there is a hit, we always predict taken, until we have a mispeculation.

6.4 Exiting From a Trace

We switch back to in-order mode when we encounter any one of the following conditions: a branch misprediction, an exception, a memory order violation, or an exit from a trace region. When this occurs, the register map has to be restored to the state of the last branch is committed, and fetch is switched to in-order instruction cache fetch. In this case, the register map will most likely reference definitions in the last non-speculative register window, and new definitions will use registers in the RW-0 as described in Section 4.3.2.

6.4.1 Software Pipelined Loop Trail Off

An interesting feature of our design is that it obviates the need for an Epilog for the converged software pipelined schedules. We keep executing out of the trace cache until we mispredict a branch in the converged schedule or mispredict the branch signaling the end of the loop. Extra instructions from non-existent future iterations executed because of software pipelining do not update non-speculative state. Their register values will be written to a future register window that will not be used. At the time of misprediction we return to in-order mode executing the fall-through path of the loop.

6.4.2 Memory Order Checking

We make sure that the in-order core checks for memory aliasing, and stops using the trace schedule if a violation is detected. This is achieved by implementing a small disambiguation checker that is indexed with the effective address of load and store operations. Note that we only need to keep track of loads and stores issued within a sliding window of 6 conditional branches (i.e. as far back as we allow speculation to occur). On an address match between a load and a store, the checker can determine the order they should have occurred from their iteration numbers. This tells the checker if the memory instructions belong to different iterations of the loop, or to the same iteration of the loop. Upon detecting a violation, the checker forces the termination of trace mode and recovers the register map to that after the branch preceding the earlier of the two conflicting instructions. Execution resumes in in-order mode after this.

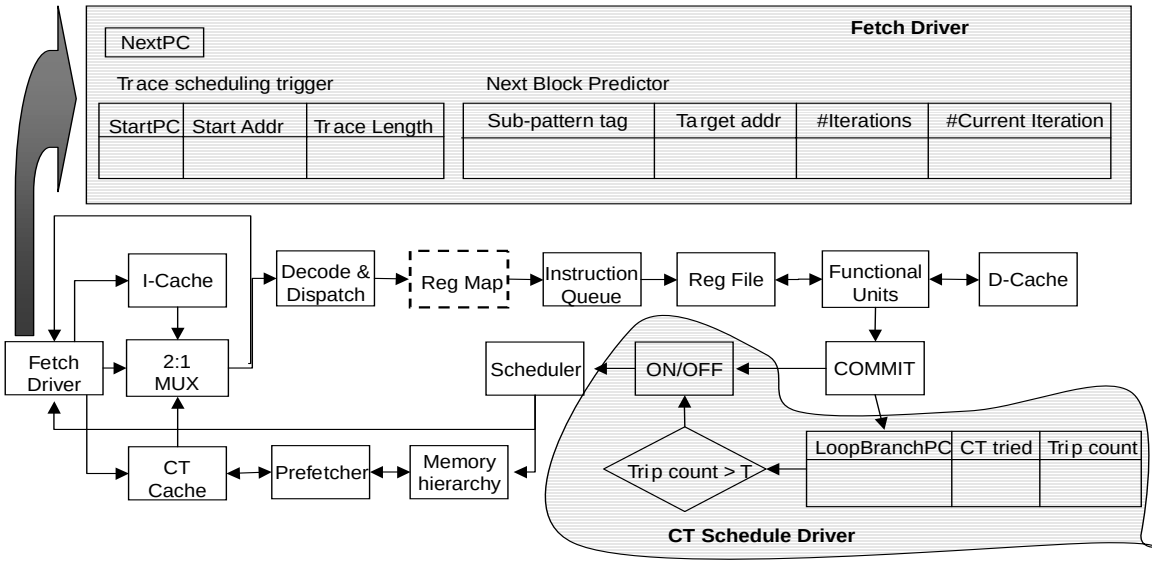

Figure 3: In-order processor with additional hardware components added to create and use converged schedules. Fetch Driver and CT-Schedule driver are shown in detail.

7 Methodology

We make use of detailed cycle accurate simulation in this study to analyze our approach. The simulator used was derived from the SimpleScalar/Alpha 3.0 tool set [8], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of a dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

To perform our evaluation we collected results for selected programs from SPEC 2000 integer and floating point benchmarks. All programs were compiled on a DEC Alpha AXP-21164 processor using the DEC FORTRAN, C or C++ compilers under OSF/1 V4.0 operating system using full compiler optimizations (`-O4 -ifo`). At `-O4 -ifo`, these DEC compilers do very aggressive optimizations like procedure inlining, loop unrolling etc., at the cost of increased code size. For the programs where loop unrolling was disabled, we used `-O4 -ifo -unroll 1` flags.

To attain reasonable simulation times, we executed a single 100 million instruction simulation point for each program. The simulation points to achieve representative results of the whole program were obtained by using the SimPoint [24] tool.

7.1 Baseline Architecture

Our baseline simulation configuration models a current in-order processor microarchitecture. We have selected the parameters to capture underlying trends in microarchitecture design. The processor may fetch and issue up to 4 instructions in-order per cycle. It has a 64 entry commit buffer with a 32 entry load/store buffer.

In the baseline architecture, there is an 8 cycle minimum branch mis-prediction penalty. The processor has 4 integer ALU units, 1-load/store unit, 1-FP adder, 1-integer MULT, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

We rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. The L1 instruction cache is a 16K 4-way associative cache with 32-byte lines. The baseline results are run with a 16K 4-way associative data cache with 32-byte lines. A 128K unified 4-way L2 cache is also simulated with 64-byte lines. The L2 cache has a latency of 12 cycles. The main memory has an access time of 200 cycles. The L1 to L2 bus can support up to 8 bytes per processor cycle whereas the L2 to memory bus can support 4 bytes per cycle. The instruction cache makes use of next-line prefetching [25] to exploit the spatial locality.

8 Results

This section first presents some statistics on each trace generated by the converged trace scheduling algorithm. Then we compare the amount of speedup obtained by our converged trace schedules to that of single pass trace scheduling approach.

8.1 Converged Trace Schedule Statistics

Table 1 presents statistics gathered from the Converged Trace Schedules (CTS) formed. For example, the Table shows that the 1st converged loop in `galgel` consisted of 6 trace blocks, the trace blocks had a recurring pattern of $(ABCD(E)^2)*$, and the loop covered 20% of the executed instructions for the given input. The original loop consisted of only 17 static instructions, and the final converged trace schedule had 96 static instructions in its prolog and 102 static instructions in the kernel. To find the converged schedule, our scheduling algorithm had to schedule only 2,142 instructions.

Program	IPC	# TB	# Static Insts	Prolog length	Kernel length	% Coverage	# Dyn Inst	Patterns
applu-ref	0.38	1	452	902	452	43.43	15368	$(A)*$
art-110	0.12	6	88	499	528	16.5	5632	$(ABCDEF)*$
		6	88	478	528	12.6	5456	$(ABCDEF)*$
		6	14	74	42	17.2	448	$(ABC^2D^2E)*$
bzip-graphic	0.888	5	24	56	108	59.02	1968	$(ABC(D)^2)*$
crafty-ref	0.602							none
galgel-ref	0.623	6	17	96	87	20.4	2142	$(ABCD(E)^2)*$
		11	16	90	107	19.7	1600	$(AB(C)^4AB(C)^3)*$
gap-ref	0.601	3	9	48	19	12.9	648	$(A^2B)*$
gzip-random	0.643	2	218	416	562	33.2	9156	$(AB^2CD)*$
lucas-ref	0.602	3	150	447	450	98.3	7500	$(ABC)*$
mcf-ref	0.056	5	31	73	92	91.9	820	$(AB(C)^2D)*$
mgrid-ref	0.568	6	73	432	396	36.7	4380	$(AB(C)^2DE)*$
		6	72	426	391	7.9	4320	$(AB(C)^2DE)*$
		6	16	90	85	6.1	960	$(AB(C)^2DE)*$
		6	8	42	41	2.0	480	$(AB(C)^2DE)*$
perl-diffmail	0.681							none
swim-ref	0.48	2	462	922	924	93.2	19404	$(AB)*$
twolf-ref	0.365							none
wupwise-ref	0.801	6	87	218	273	23.3	2472	$(AB^2CD^2E)*$

Table 1: Statistics describing the generated converged traces schedules (CTS). Results are shown for the number of trace blocks (TB) in the schedule, static number of instructions in the original loop, prolog and kernel lengths of trace formed, percentage of executed instructions consumed from the trace, dynamic number of instructions sent to the scheduler, trace block pattern of the trace. If more than one loop for a program are trace scheduled they are shown on separate lines.

Several observations can be made from the table. First, one can observe that we are able to identify complex patterns in the converged traces through our string matching algorithm. It can also be noted that for some of the traces, the kernel length is not a multiple of the corresponding static size of the loop. This is because we don't include paths that are not taken in the schedule and we perform inlining. In addition, we compactly represent the kernel through our compression algorithm. Another trend that is striking is the number of static instructions in the original loops. Some integer programs like `crafty`, don't have loops with high trip counts and hence they don't have any converged trace schedules.

8.2 Single Trace Block Scheduling

In addition to our converged scheduling algorithm, we also implemented a baseline algorithm that is close to traditional trace scheduling and the algorithm used in the DIF cache [18] implementation mentioned in Section 2. We call this the Trace Scheduling (TS) Algorithm. To build these traces, we use an out-of-order scheduler to generate single path trace schedules. But instead of generating schedules for multiple iterations and finding patterns in those schedules, here the final schedule is formed by scheduling only one trace block (that is, a trace through one loop iteration). In addition, we limit the scheduling window size is limited to 64 instructions.

Figure 4 shows the percentage of executed instructions that came from the different schedules formed for using Trace Scheduling (TS), Converged Trace Schedules (CTS), and CTS with Pattern Trace Schedules (PTS) as described in Section 4.5. PTS represents the frequent trace block patterns found for loops that did not converge. The results show that even with a few traces we are able to cover 52% of execution of these programs on average.

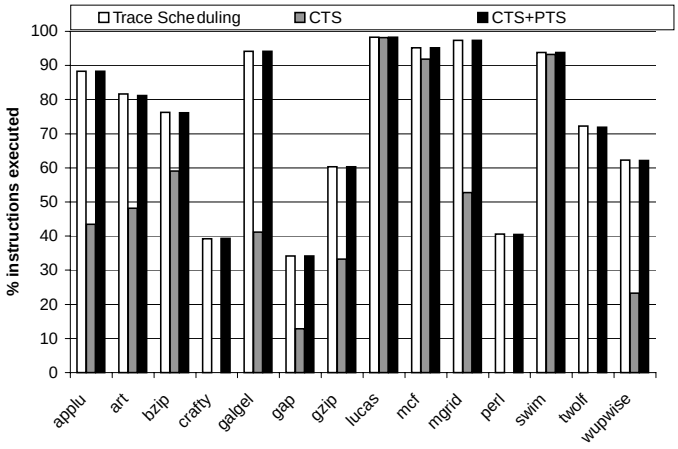

Figure 4: Percentage coverage of program execution from the various trace scheduling techniques.

8.3 Non-Loop Unrolled Results

Loop unrolling is an optimization done by the compiler for efficient scheduling. Since we concentrate on generating efficient schedules for loops, here we show how our converged scheduling technique is able to achieve the same performance benefit as compiler loop unrolling. These binaries for this one result were compiled with -O4, but with loop unrolling turned off.

Figure 5 shows the IPC for in-order execution of binaries generated without loop unrolling (W/O LU-Inorder) and compares it with trace scheduling (W/O LU-TS), converged CTS and also CTS with PTS combined. It can be seen that converged CTS does a good job in achieving similar performance as loop unrolling performed by the compiler. For `gzip` and `wupwise`, CTS does not provide a lot of benefit because of the irregular control flow in these applications. In comparison, `applu` achieves nearly 19% speedup using converged CTS alone even though the schedule covers only 43.3% of instructions executed.

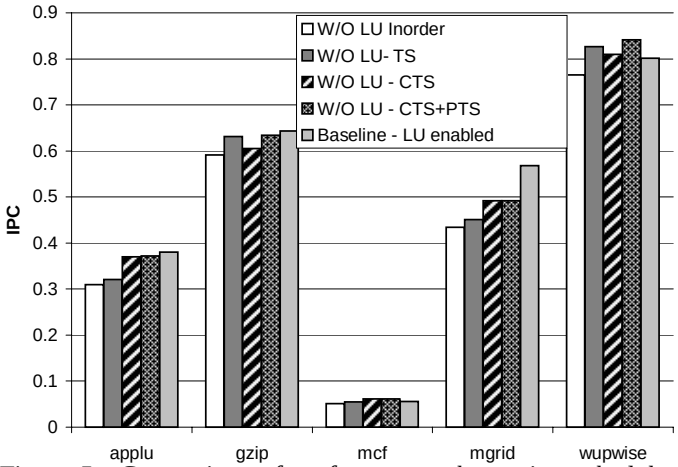

Figure 5: Comparison of performance when using scheduler for binaries generated with loop unrolling (LU) compiler optimization disabled (W/O Loop unrolling). Converged CTS is able to achieve almost same performance as doing loop unrolling in compiler. The last bar shows the baseline results with loop unrolling turned on.

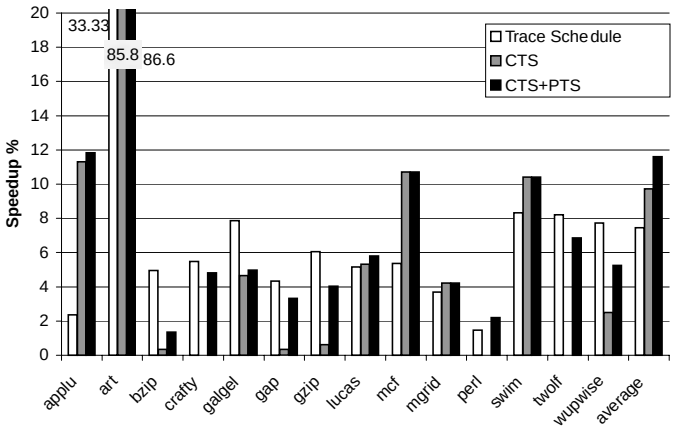

Figure 6: Percent speedup over baseline in-order architecture for serial thread model.

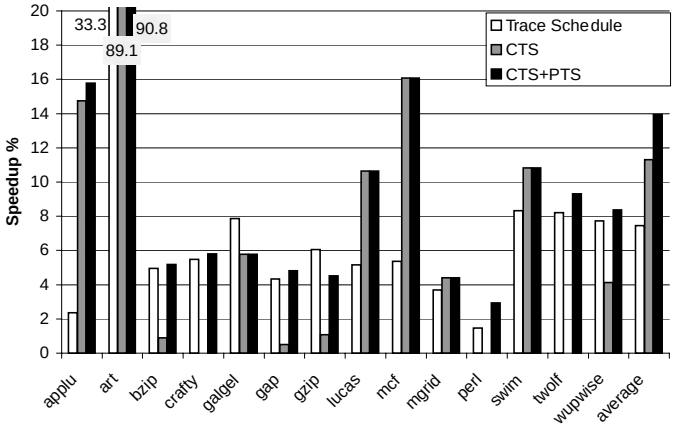

Figure 7: Percent speedup over baseline in-order architecture for parallel thread model.

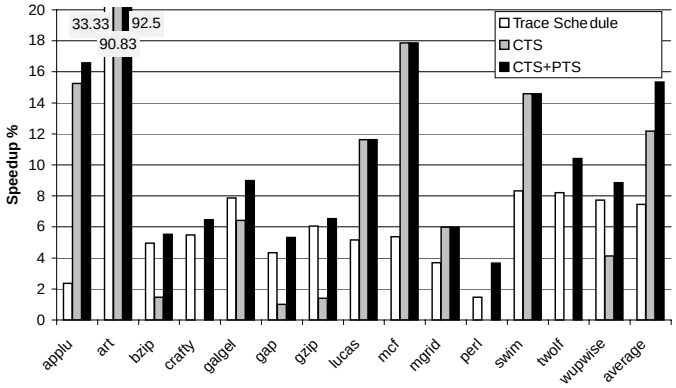

Figure 8: Percent speedup over baseline in-order architecture for coprocessor model.

This can be compares favorably to the 4% improvement due to single pass trace scheduling even though it covers nearly 90% of instructions executed as shown in Figure 4.

8.4 Loop Unrolled Results

We now report results using binaries generated with -O4 with loop unrolling optimization enabled, and speedups are reported over these baseline loop unrolled binaries. Figures 6, 7, and 8 show the percent speedup for serial, parallel and coprocessor models that were described in Section 5.

Remember that all of the results are for an in-order processor using instructions from either the I-cache or the CT-cache. The speedups are calculated over the baseline in-order architecture. In case of the serial model, the scheduling thread competes with the main thread and hence incurs overhead. For baseline single pass scheduling, this overhead is lesser than CTS and PTS schedules. In spite of this, the converged scheduling algorithm for the CTS plus PTS schedules obtains 12% speedup on an average when compared to that of 8% for the baseline TS. This gap further widens for parallel and coprocessor implementations, as performance of converged scheduling improves to 14% and 16% respectively.

The results in Figure 8 show that we achieve as much as 91% speedup from CTS schedules only for `art` in the coprocessor implementation. This performance is significant especially when one considers the fact that only 48% of the instructions (not cycles) executed were from CTS schedules for `art`. We achieve this through efficient scheduling of instructions, especially load instructions. Also from Figure 8 we show that programs like `perl`, for which a converged trace schedule was not created, there is a 4% performance improvement from the PTS schedules created. Here, the pattern trace schedules formed performed better than the single Trace Schedule algorithm, which yields under 2% performance improvement for `perl`.

8.5 Sensitivity Analysis

We did experiments to evaluate the heuristics used in the ILP scheduler. For the results reported in the previous sections, we limited the number of branches that an instruction can be speculated above to be 6. When we increased this to 10, we observed a performance improvement for some benchmarks like `galgel` and `mcf` by about 5%. But this came at the cost of increased

kernel and prolog lengths and also longer time for convergence. Other benchmarks like `applu`, `swim` and `lucas` didn't benefit from this increased speculation as the number of instructions between two branches in these benchmarks was very high and as a result, the number of instructions spanning 6 branches was already greater than our instruction window size of 500.

9 Conclusions

In this paper, we focused on generating software pipelined schedules called Converged Trace Schedules (CTS). Our algorithm aggressively schedules the in-order sequence of instructions resulting from the execution of multiple iterations of the candidate loop. This scheduling is done assuming architectural support for efficient register renaming and speculation recovery. The resulting schedules are divided into trace blocks, which are then converted into a string of IDs. The string of IDs are then sent to a recurring pattern finding algorithm to recognize the kernel of converged trace schedule.

We evaluate the efficiency of this algorithm by using it in a dynamic optimizer for an in-order architecture. For the programs that we examined, less than 0.02% of the total number of instructions in a program's execution were used for generating the converged CTS schedules. This 0.02% is with respect to the 100 million instructions we simulated. We expect the overhead to be significantly smaller when it is amortized over the complete execution of the program, which is typically on the order of 100s of billions of instructions. The results show that creating these highly specialized converged kernels achieved an average of 12-16% speedup on the collection of programs we examined depending on the implementation.

In this work we concentrated on using string matching to create converged trace schedules for loops on-the-fly. These schedules were tailored to the latencies and other characteristics of the base architecture. It is possible to create these traces of schedules offline by a simple, but detailed, simulator for the baseline architecture. Then a compiler could include these trace schedules in the binary, and during execution they can be loaded directly into the converged trace cache and used. The processor can then use these schedules, provided it has the additional hardware features supporting speculation as described in this paper. Including these specialized schedules into a compiled binary and directly fetching them into a trace cache is a topic of future research.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF Grant No. CCR-0105743, NSF Grant No. CCR-0311683, and a grant from Intel.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. pages 274–290, 1991.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[4] V. H. Allan, M. Rajagopalan, and R. M. Lee. Software pipelining: Petri net pacemaker. pages 15–26, 1993.

[5] B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. In *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, pages 196–207, May 1999.

[6] B. Black and J. Shen. Scalable register renaming via the quack register file. Technical Report CMuArt 00-1, Carnegie Mellon University, April 2000.

[7] B. Black and J. P. Shen. Turboscalar: A high frequency, high ipc microarchitecture. In *Workshop on Complexity-Effective Design, International Symposium on Computer Architecture*, June 2000.

[8] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[9] Y. Chou and J. P. Shen. Instruction path coprocessors. In *The 27th Annual International Symposium on Computer Architecture*, pages 270–281, June 2000.

[10] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. pages 69–79, 1987.

[11] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[12] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a vliw architecture. pages 213–229, 1989.

[13] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *27th International Symposium on Microarchitecture*, pages 162–171, December 1994.

[14] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *30th International Symposium on Microarchitecture*, pages 24–33, December 1997.

[15] Daniel H. Friendly, Sanjay J. Patel, and Yale N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *International Symposium on Microarchitecture*, pages 173–181, December 1998.

[16] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 278–289, June 1993.

[17] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. pages 258–267, 1988.

[18] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[19] E. M. Nystrom, R. D. Barnes, M. C. Merten, and W. W. Hwu. Code reordering and speculation support for dynamic optimization systems. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2):193–204, 1999.

[21] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. pages 183–197, 1981.

[22] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th International Symposium on Microarchitecture*, December 1994.

[23] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, December 1996.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, October 2002.

[25] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.