Copyright 1997 IEEE. Published in the Proceedings of ICPP'97, August 11-15, 1997 in Bloomingdale, Illinois. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

Multiscalar Execution along a Single Flow of Control

Krishna K. Sundararaman
Intel Corporation
1900 Prairie City Road
Folsom
CA 95630, USA
ksundara@pcocd2.intel.com

Manoj Franklin
Dept. of Electrical and Computer Engineering
Clemson University
221 Riggs Hall
Clemson, SC 29634-0915, USA
mfrankl@blessing.ces.clemson.edu

Abstract

The multiscalar processing model extracts instruction level parallelism from ordinary programs by splitting the program into smaller, possibly dependent, tasks, and parallelly executing multiple tasks using multiple execution units. Past work had advocated pursuing multiple flows of control in the multiscalar processor. We first illustrate the problems involved in pursuing multiple flows of control. We then discuss a methodology to obtain good performance from multiple tasks extracted from a single line of control. We also present the results of simulation studies that verify the potential of this method. These results, obtained with a set of SPEC92 benchmarks, show better issue rates when a single line of control is pursued in the multiscalar processor. The primary reason for this improvement is the ability to have better load balancing among the execution units.

1 Introduction

Recent advances in VLSI technology have enabled millions of transistors to be placed in a single chip. Current processors incorporate about 15 million transistors in a chip, and technology projections predict this number to reach about 100 million by A.D. 2000 [7]. One way these resources are being used is for incorporating performance-enhancing hardware features, targeted towards exploiting different kinds of parallelism. A processor of this kind that has good potential for exploiting instruction-level parallelism (ILP) is the *multiscalar processor* [4] [5] [14]. It exploits ILP by parallelly executing multiple blocks of code (called *tasks*) in multiple, decentralized execution units that are connected together using a ring-type network. A major tenet of the multiscalar processing paradigm is its ability to parallelly execute tasks that have control dependences as well as data dependences between them.

The primary goal of this research is to investigate the potential of following a single line of control in the multiscalar processor. Why study single line of control, at a time when multiple flows of control (MFC) is being touted to replace it soon [2] [8] [13]? First, branch prediction accuracies have improved significantly in recent years, thanks to 2-level prediction schemes [15], correlation-based prediction schemes [11], and hybrid prediction schemes [1]. These prediction schemes have prediction accuracies averaging over 95% for non-numeric programs, pushing the average distance between mispredictions to 100+ instructions. Second, the debilitating effect of hard-to-predict (short) branches can be overcome to a large extent by using predicated execution [9]. Third, today it is possible to execute branches out-of-order [10] while pursuing a single flow of control. Lastly, runtime scheduling is a lot easier on the hardware when a single flow of control is pursued. There is a significant amount of complexity involved in supporting multiple flows of control, as evidenced by the descriptions given in [2] [4]. Consequently, it becomes important to evaluate the effectiveness of following a single line of control in the multiscalar processor.

The rest of this paper is organized as follows. Section 2 provides background information on the multiscalar processor. Section 3 provides a fresh look at pursuing speculative execution along multiple flows of control, and details the hurdles involved in implementing multiple flows of control in the multiscalar processor. Section 4 describes a methodology for executing multiple threads from a single flow of control (SFC) in the multiscalar processor. Section 5 presents the performance results of SFC-based multiscalar execution and MFC-based multiscalar execution, obtained through detailed simulation studies. These results show that pursuing a single line of control within tasks provides better performance in the multiscalar processor. Section 6 presents the conclusions.

2 Background

2.1 Multiscalar Processor

The multiscalar processor [4] [5] [14] employs a new execution model for extracting large quantities of ILP from ordinary programs. Figure 1 gives the block diagram of a multiscalar processor. It consists of multiple execution units (EUs) that are connected by a unidirectional ring-type network. This provision, along with hardware head (H) and tail (T) pointers, imposes a sequential order among the units, the head pointer indicating the oldest active EU.

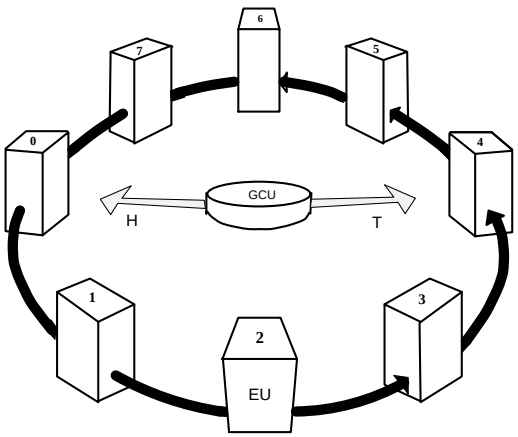

Figure 1: Block Diagram of an 8-Unit Multiscalar Processor

A program executes on the multiscalar processor as follows. Each cycle, if the tail EU is idle, the global control unit (GCU) predicts the next task in the dynamic instruction stream, and invokes it on the tail EU; a task is a subgraph of the control flow graph (CFG) of the executed program. After invocation, the tail pointer is advanced, and the invocation process continues at the new tail in the next cycle. Thus, the GCU steps through the CFG, distributing tasks (speculatively) to the multiscalar's EUs. The tasks being executed in parallel can have both control dependences and data dependences between them. Each active EU executes the instructions in its task, maintaining the appearance of sequential semantics within the tasks and between the tasks. When the head EU completes its task, the instructions in its task are committed, and the head pointer is advanced, causing that EU to become idle. When a task misprediction is detected, all EUs between the incorrect speculation point and the tail are discarded in what is known as a *squash*. The tail pointer is adjusted to point to the EU at the incorrect speculation point, and the invocation process resumes along the correct path.

The parallel EUs support a single, conventional set of logical registers as well as a conventionally addressed memory system. Register results are dynamically routed among the EUs with the help of a unidirectional ring-like network. Memory references may be executed out of order, speculatively, without knowledge of preceding loads or stores. Memory address hazards are resolved dynamically, many in parallel, with a hardware address resolution mechanism [6].

2.2 Example

Figure 2 ahows a CFG consisting of three basic blocks A, B, and C, with block B *control-dependent* on the conditional branch in A, and block C *control-independent* of the branch. Two multiscalar tasks, T0 and T1, have been formed; the first task contains a diamond-shaped control structure, and the second task is control-independent of the first task. At run-

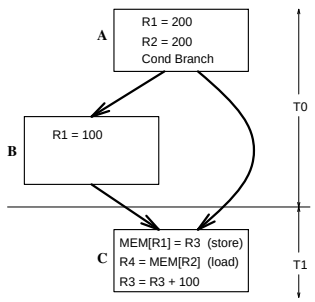

Figure 2: Example Control Flow Graph and Code

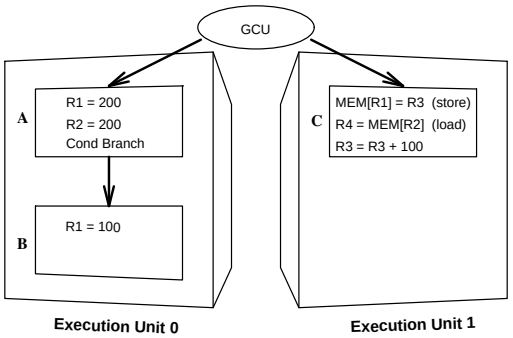

Figure 3: Multiscalar Execution of Example Code

time, the multiscalar processor executes multiple dynamic tasks in the EUs, as shown in Figure 3, effectively establishing a large dynamic window of tasks. Notice that no prediction is done at the "internal" branch point of task T0 in establishing the overall dynamic window of tasks, and this can potentially provide higher prediction accuracies at the task level.

3 Multiple Flows of Control

A typical program encompasses many alternative paths, out of which only one is taken when the program is executed with a specific set of inputs. In the early days, a processor would strictly follow this *single flow of control* (SFC); when a conditional branch is encountered, it would wait until the branch is resolved. With the introduction of speculative execution, the methodology of pursuing an SFC took a slightly different turn. The most likely path was speculated by predicting the outcome of conditional branches, and this path was pursued, with provisions to recover and pursue the correct path when a prediction was found to be incorrect. Today's processors typically do out-of-order execution within the speculated path, without giving any considerations to the control dependences within the path [13]. Notice that it is possible to use multiple sequencers to fetch different portions of the speculated path, once the path is determined by means of a predictor. For instance, once the predictor has predicted the conditional branch in basic block A to be fall-through (i.e., the predictor has decided the path ABC), then three separate sequencers can be used to fetch the instructions belonging to blocks A, B, and C, and it would still amount to speculative execution along an SFC.

When speculating along an SFC, recovery from a misprediction involves squashing all of the subsequent instructions, including those that are not control-dependent on the mispredicted branch (which would be fetched and executed again). For instance, when a misprediction for the branch in block A is detected, block C has to be squashed too, although it has to be fetched and executed again in the alternate path, AC. Recent papers have advocated following *multiple flows of control (MFC)* to avoid this inefficiency [2] [8]. Processors that follow MFC have knowledge about the control dependences within the code that it is currently executing, and use that knowledge to selectively squash only the portion of code that is control dependent on the mispredicted branch.

In the execution depicted in Figure 3, the multiscalar processor is pursuing MFC because it is executing in parallel two control-independent tasks. The first task encompasses two alternate flows of control. Irrespective of which one of these is taken at run-time, the task executed in EU 1 is the same, namely T1. If the conditional branch in task T0 is found to be mispredicted, no squashing is done in EU 1 and subsequent EUs. It is this aspect of multiscalar processor—executing MFC—that we investigate in this paper.

3.1 Parallelism Limits

Previous studies [8] had found that without doing *speculative execution within multiple flows of control*, the amount of parallelism available with multiple flows of control is quite limited. That is, when a control sequencer encounters a conditional branch, it must predict the branch outcome and proceed, instead of waiting for the branch to be resolved[1]. Although the upper limit of theoretically exploitable parallelism was found to be high when both speculation and multiple flows of control were pursued, a major assumption had been used in that study concerning potential data dependences from control-dependent portions of code to control-independent portions of code. We shall illustrate this with our example CFG.

In this CFG, although the store instruction `MEM[R1] = R3` in block C is control-independent of the outcome of the branch in block A, the store address is dependent on the branch's outcome. Assume that the branch was incorrectly predicted to be *fall-through*, and that both `R1 = 100` and `MEM[R1] = R3` were executed before detecting this misprediction. When the misprediction is detected, the store instruction has to be re-executed (because of the change in R1) even though it is control-independent of the mispredicted branch, and the studies in [8] do not include such re-executions. Similarly, if the branch is incorrectly predicted to be *taken*, the store has to be re-executed when this midprediction is detected owing to the store address' data dependence on the `R1 = 100` instruction that is newly introduced into the execution path. If the effect of such re-executions is considered, the difference in parallelism limits between speculative execution along an SFC and speculative execution along MFC is likely to shrink.

Furthermore, the studies in [8] had obtained average branch prediction accuracies of about 90% for non-numeric benchmarks. After that study was done in 1992, branch prediction accuracies have improved substantially. Today, a variety of high-accuracy branch prediction schemes, such as 2-level predictors [15], correlation-based predictors [11], and hybrid predictors [1] yield more than 95% branch prediction accuracy for non-numeric programs. Another blow to the imposition of branches was the advent of predicated execution [9], which can be effectively used to eliminate (short) hard-to-predict branches. As branch prediction accuracies increase, the benefits of MFC nat-

[1] It is certainly possible to pursue both outcomes of the branch simultaneously; however, that would lead to exponential growth in hardware requirements if many predicted-but-unresolved branches are present in the instruction window.

urally tend to decrease. Finally, the studies in [8] had assumed that SFC-based processors need to execute branches in sequential order. This is overly restrictive, considering that today's SFC-based processors execute branches out-of-order (e.g. MIPS R10000).

3.2 Implementation Hurdles

We saw that speculative execution along MFC offers a performance advantage only when there are mispredictions in which subsequent portions of the hardware window contain code that is both control-independent of the mispredicted branch and data-independent of both the squashed code and the newly introduced code. Next, let us look at some of the implementation issues involved in allowing the hardware to pursue speculative execution along multiple flows of control.

The first implementation hurdle relates to determining data dependence violations at times of recovery from mispredictions. Data dependence violations occurring through memory are naturally more difficult to determine than those occurring through registers. Consider again the example code of Figure 2. Assume that the conditional branch of task T0 was mispredicted to be taken, and that both the store and the load instructions of task T1 have been already executed (to address 200). After recovering from the misprediction, the `R1 = 100` instruction is newly introduced into the instruction window. It is obvious that the store instruction has to be re-executed (because its address changes due to the change in R1's value). What is not so obvious is that the load instruction should also be re-executed because there was a data dependence between the store and the load in the incorrectly speculated path (in which the store address and load address are same), whereas there is no data dependence between them in the new path (in which their addresses are different). Thus, the memory dependence checking hardware has to re-check for memory dependences when a store instruction is re-executed, squashed, or introduced into the middle of the instruction window.

The second hurdle relates to implementing *selective re-execution* of instructions whose data dependences have been violated due to squashing of code or introduction of new code. For example, the store instruction and load instruction in the above example have to be selectively re-executed, whereas the `R3 = R3 + 100` instruction need not be re-executed. The importance of selective re-execution cannot be overemphasized here; if all of the instructions after a mispredicted branch are squashed whenever a data dependence is violated, then the benefit obtained by pursuing MFC will inevitably be lost.

The third implementation hurdle relates to bringing in additional code into an EU at times of recovery from incorrect speculation. For instance, when recovery is carried out in the previous example code, an additional instruction has to be brought into EU 0. One option to do this is to leave space in the EU for the longer path whenever a control speculation is made; however, that could result in underutilization of the EU's instruction window structure.

3.3 Performance Hurdles

Let us next look at the performance hurdles involved in pursuing MFC. First of all, our recent experiments with state-of-the-art branch predictors show only a factor of 2 improvement in available parallelism for MFC over SFC even when an infinite size instruction window is used. With instruction windows of size less than 100, there is only modest difference in available parallelism. Secondly, the additional available parallelism may not easily translate to performance in multiscalar processors! The main reason for this is the difficulty in providing *load balancing* among the EUs while pursuing MFC. In multiscalar execution, because tasks must be committed in order, cycles may be lost if tasks are not (roughly) of the same size in terms of dynamic instructions. Consider the scenario depicted in Figure 4, where a big task has been assigned to EU 0, and a small task has been assigned to EU 1. Even though EU 1 is able to complete its

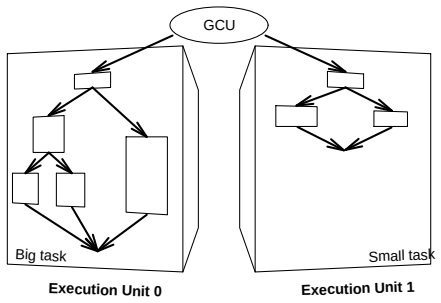

Figure 4: Illustrating Load Balancing's Importance in Multiscalar Processors

small task quickly, it has to wait for the big task in EU 0 to be completed and committed, before committing its task. Lastly, the increased hardware complexity of implementing MFC could lead to increased cycle time. It is difficult to quantify this increase without doing a detailed hardware implementation, which is beyond the scope of this paper.

4 SFC-Based Execution

The existence of the hurdles detailed above prompted us to investigate the potential of pursuing a single line of control in the multiscalar processor. A simple way to do this is to predict the outcome of each conditional branch. However, if predictions are done at the branch level, at the rate of one prediction per clock cycle, then the multiscalar processor cannot advance more than a basic block at a time. Therefore, we use higher-level predictors like the one proposed in [3]. The basic idea is to form tree-like tasks, containing multiple paths, from the dynamic CFG so that a single path can be predicted from the tree, and pursued.

4.1 Tree-Like Tasks

Figure 5 shows the generic structure of a tree-like task of depth 3. Each depth consists of one or more straightline blocks of code. All blocks except the

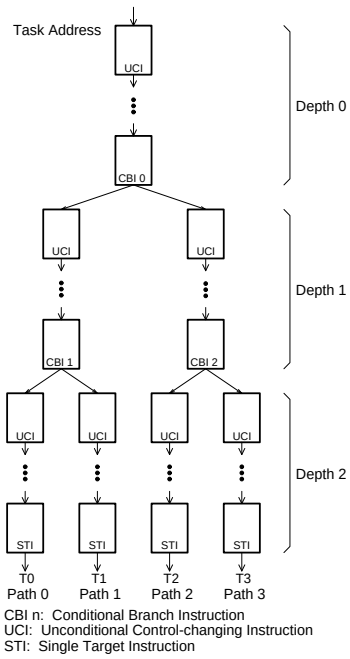

Figure 5: Generic Structure of a Tree-like Task that Encompasses 4 Tree-Paths

last one at each depth are terminated by an unconditional control-changing instruction (UCI), such as a procedure call or an unconditional branch. The last blocks of depths 0 and 1 are terminated by conditional branch instructions (CBI), whereas the last blocks of depth 2 are terminated only by single-target instructions (STI), such as unconditional branches, procedure calls, and non-control-changing instructions. Such a definition allows up to 4 paths (called *tree-paths*) to be encompassed in a task, and up to 2 conditional branches and any number of single-target control-changing instructions in each tree-path. Thus, a tree-path may have a call to a function, and also a return from the function back to the calling place, thereby inlining the function automatically. The task has 4 targets {T0, T1, T2, T3}, one corresponding to each tree-path. The 4 tree-paths are encoded {0, 1, 2, 3}.

Each tree-path is a snapshot, or portion of the trace, of a possible dynamic instruction sequence. The length of a tree-path is restricted by two factors—by the maximum number of instructions it is allowed to have and by the maximum number of conditional branches it is allowed to have. The former limit is chosen to provide some sort of *load balancing* among the multiple EUs in the multiscalar processor, *i.e.*, to have tree-paths of roughly the same length. The latter limit is chosen so that the task predictor needs to do only an 1-out-of-4 prediction while selecting a tree-path through a task, which can be done with good accuracy [3]. A tree-path is specified by its starting address and a 2-bit number that encodes the outcomes for the 2 conditional branches (if any) encompassed within the tree-path. The tree-paths could be built up as the program executes, or could be generated by a post-compilation phase of the compiler.

With tree-like tasks, multiscalar execution proceeds as follows. The GCU has the address of the next task to be allocated to the tail EU. It makes an 1-out-of-4 prediction to select one of the four tree-paths contained in the task, and conveys the task address and the selected tree-path number to the EU. Once the GCU selects a path through a task, the next task is automatically decided because the start address of the next task is an attribute of the selected tree-path.

An incorrect tree-path prediction will be detected when the appropriate conditional branch in the tree-path is executed. When a misprediction is detected, recovery actions are taken by squashing the tasks in the subsequent EUs. In addition, the tree-path containing the incorrect prediction is also squashed, and the correct tree-path is invoked in the same EU. If no run-time scheduling is done within EUs, then the part of this tree-path before the incorrect branch need not be squashed, because it will be common to the old (incorrect) path as well as the new (correct) tree-path.

4.2 Icache Organization

Next, let us consider the issue of storing tree-like tasks in the first level (L1) local icaches present in each

EU. Although a conventional cache organization, with some added information about the control changes in a task, will suffice for storing the instructions of a task, this may not be the best approach for storing tree-like tasks. We propose to use an alternative organization, similar in spirit to the trace cache proposed in [12]. In this organization, instructions of a tree-path are stored together in the icache in their dynamic trace order, and not in their compiled order. The cost of this approach is potential redundancy among the icache lines due to multiple tree-paths having same instructions.

4.3 Advantages

Perhaps the biggest advantage of pursuing a single flow of control in the multiscalar processor is *hardware simplicity*; none of the implementation complexities listed in section 3.2 are present. A consequence of following SFC-based execution is that run-time reordering of instructions within a task (if deemed necessary) becomes easier to do in hardware. Another advantage of SFC-based execution is the ease of doing *load balancing* among the EUs. In SFC-based execution, because the length of a tree-path is not dependent on run-time behavior, the length is known at tree-path formation time, which makes it easy to form tree-paths that are roughly of the same size.

Another aspect unique to the multiscalar style of execution is that if the "internal" branches in a task are predicted locally by an EU, then the prediction accuracies are likely to be poor, because each EU has only partial history information about each branch. Thus, the speculative path followed within a task by an EU when following MFC is likely to be less accurate than the path followed by an EU that takes its cues from a GCU that makes more accurate tree-path predictions at the global level, using complete history.

5 Experiments and Results

5.1 Experimental Framework

Our simulator uses the MIPS-I instruction set. Benchmark programs are compiled to MIPS-I executables using the MIPS compiler. A task decider program takes the MIPS-I executables, and decides the tasks. The executable and the task information are then fed to a multiscalar simulator. The simulator models all aspects of the multiscalar architecture accurately and in detail and gives the exact time to execute the benchmark program on a real multiscalar processor. The simulator accepts executable images of programs (along with task information), and simulates their execution; it is not trace driven. The simulator also incorporates a mini-operating system to handle the system calls made by the simulated program. The code executed during the system calls are not taken into account in the performance results. Each of the benchmarks is simulated for 100 million instructions.

For benchmarks, we used a set of programs from the SPEC92 benchmark suite. For measuring performance, we use *useful instruction completion rate (UICR)* as the metric. Notice that nops and squashed instructions are not counted while calculating UICR.

There are many parameters that can be varied while conducting the simulation studies. We fixed the following parameters so that they do not distract us from the tradeoffs under investigation.

- Each tree-path is restricted to 16 instructions.
- The number of execution units is fixed at 8.
- Dynamic scheduling is performed in each EU, and up to 4 instructions per cycle are executed from each active EU.
- Each execution unit has a 4Kwords local icache, with a miss latency of **4 cycles**.
- 100% hit ratio is assumed for the global icache.
- The common, L1 dcache is 64Kbytes, direct-mapped, and has an access latency of **2 cycles**.

5.2 MFC Scheme Simulated

For comparison purposes, we also simulated the traditional multiscalar processor that pursues speculative execution along MFC. In this scheme, the task decider forms tasks by identifying from the CFG subgraphs that are control-independent of each other. However, many a times, such tasks have too much variance in size. In order to reduce this variance, we prune the number of instructions in a task to a particular value (32 in our experiments). When a task is pruned, it will most likely have multiple, control-dependent, target PC values. Because the instructions in a task form a subgraph of the CFG, and may not be from a contiguous portion of the compiled code, there is a plethora of ways the task decider can do the pruning. To give the best treatment to the MFC case, up to 4 possible tasks are formed for each possible task start address. Each of these tasks is restricted to 32 instructions, but may have arbitrary number of conditional branches and flows of control within it.

If the latest task allocated by the GCU to an EU has multiple targets, then the GCU makes a prediction to determine the most likely successor target. Another dynamic predictor within the GCU selects one out of the 4 tasks starting at that target address, and assigns it to the tail EU. Within each EU, further speculation is done to select the most likely path through the task that is currently allocated to it. The branch predictions within each task are carried out by means of a global, 3-bit saturating counter-based predictor. Notice that a pattern-based (2-level) predictor does not work well for intra-task predictions, because the intra-task predictions in the multiscalar processor are not done as per the program's sequential execution order.

Let us explain how the implementation issues mentioned in section 3.2 are addressed in this simulation study. When an intra-task branch prediction is found to be incorrect, local recovery actions are taken, which might involve squashing some instructions and fetching new instructions. Notice, however, that control-independent instructions in the same EU or subsequent EUs are not squashed when a data dependence is found to have been violated. Instead, the affected instructions are re-executed when their correct operands become available. Store instructions that are on a locally speculated path are not executed until all conditional branches before it (in that task) are resolved. This avoids the need to remove an executed store from an incorrect path or the need to re-execute a store, thereby avoiding the associated memory data dependence checking problem. Lastly, the hardware scheduler in each EU is made bigger to accommodate the bigger task sizes that are encountered in MFC-based execution because of the presence of instructions from alternate flows of control.

5.3 Experimental Results

5.3.1 Instruction Completion Rate

Figure 6 presents the UICR obtained for an 8-unit multiscalar processor. Each benchmark has 2 bar charts; the lighter shade corresponds to MFC-based execution, and the darker shade corresponds to SFC-based execution. It can be seen that SFC-based execution using tree-like tasks has performed better than the MFC case, as expected. The main reason for this improvement is better load balancing among the EUs.

5.3.2 Average Task Size

Table 1 presents the average task sizes obtained for MFC-based execution and SFC-based execution. For

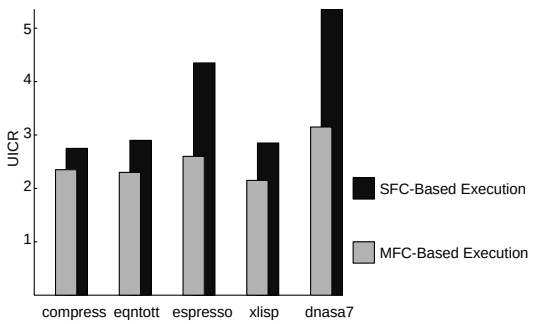

Figure 6: Useful Instruction Issue Rate with 8 EUs

Benchmark	*Average Task Size*		
	MFC-based Execution		*SFC-Based Execution*
	Overall Size	*Useful Size*	
compress	12.68	9.61	10.57
eqntott	26.33	7.94	7.79
espresso	20.16	8.37	9.69
xlisp	14.20	7.44	8.35
dnasa7	26.05	25.07	15.65

Table 1: Overall Task Size and Useful Task Size

MFC-based execution, two sets of task sizes are presented. The overall task size denotes the size of a multiscalar task, whereas the useful task size denotes the average number of instructions that are actually committed from a task. These values are different because multiple flows of control are embedded in a task. For SFC-based execution, the overall task size and the useful task size are the same. We can see from the table that there is a substantial difference between the overall task size and the useful task size for MFC-based execution, necessitating bigger instruction window in the EUs. Quite often, many of these useless instructions are fetched and executed (because of poorer local branch prediction accuracies (c.f. Table 2)), only to be discarded later. Note that it takes additional cycles to fetch and execute the useless instructions.

5.3.3 Average Task Prediction Accuracy

Table 2 presents the prediction accuracies that we obtained. The second and third columns present the task prediction accuracy (TPA) and the local branch prediction accuracy (BPA), respectively, for MFC-based execution. The last column gives the task prediction accuracy for SFC-based execution. From the table, it can be seen that the task prediction accuracies obtained with SFC-based execution are somewhat comparable to those obtained with MFC-based execution. We can also see that the intra-task branch prediction accuracies for MFC-based execution are not very high.

Benchmark	*Prediction Accuracies*		
	MFC-based Exec.		*SFC-Based Exec.*
	TPA	*BPA*	*TPA*
compress	83.21%	89.63%	84.88%
eqntott	95.64%	94.31%	91.85%
espresso	95.93%	95.06%	93.83%
xlisp	92.49%	90.48%	94.68%
dnasa7	99.06%	98.28%	99.45%

Table 2: Prediction Accuracies

6 Summary and Conclusions

Pursuing multiple flows of control, as we know it today, has benefits only when it is difficult to obtain high branch prediction accuracies and if there are not many data dependences from the control-dependent portions of a mispredicted branch to nearby control-independent portions of code. Implementing speculative execution along multiple flows of control in the hardware raises several implementation complexities. As branch prediction accuracies continue to improve, and get closer to 100%, the benefits accrued by pursuing multiple flows of control begin to fade.

This paper investigated the potential of executing multiple tasks from a single line of control (SFC-based execution) in the multiscalar processor. A simulation study of SFC-based multiscalar execution showed better instruction completion rates. Based on these results, we believe that with the continuing increases in branch prediction accuracies, SFC-based execution is more advantageous for the multiscalar processor.

Acknowledgements

This research was supported by the US National Science Foundation (NSF) under Research Initiation Award grant CCR 9410706.

References

[1] P-Y. Chang, E. Hao, and Y. N. Patt, "Alternative Implementations of Hybrid Branch Predictors," *Proc. 28th International Symposium on Microarchitecture*, 1995.

[2] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," *Proc. International Conference on Parallel Architecture and Compilation Techniques (PACT '95)*, 1995.

[3] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-like Subgraphs for Superscalar Processors," *Proc. 28th International Symposium on Microarchitecture*, pp. 258-263, 1995.

[4] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[5] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 58-67, 1992.

[6] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory Operations," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552-571, May 1996.

[7] P. P. Gelsinger, P. A. Gargini, G. H. Parker, and A. Y. C. Yu, "Microprocessors circa 2000," *IEEE Spectrum*, vol. 26, no. 10, pp. 43-47, October 1989.

[8] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 46-57, 1992.

[9] S. A. Mahlke, *et al*, "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th International Symposium on Microarchitecture*, pp. 217-227, 1994.

[10] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, Version 1.1, January 1996.

[11] S-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 76-84, 1992.

[12] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th International Symposium on Microarchitecture (MICRO-29)*, pp. 24-34, 1996.

[13] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. IEEE*, pp. 1609-1624, December 1995.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd International Symposium on Computer Architecture*, pp. 414-425, 1995.

[15] T-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.