Dynamic Thread Resizing for Speculative Multithreaded Processors

Mohamed Zahran and Manoj Franklin
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD, 20742
{mzahran, manoj}@eng.umd.edu

Abstract

There is a growing interest in the use of speculative multithreading to speed up the execution of a program. In speculative multithreading model, threads are extracted from a sequential program and are speculatively executed in parallel, without violating sequential program semantics. In order to get the best performance from this model, a highly accurate thread selection scheme is needed in order to accurately assign threads to processing elements (PEs) for parallel execution. This is done using a thread predictor that assigns threads to PEs sequentially. However, this in-order thread assignment has severe limitations. One of the limitations is when the thread predictor is unable to predict the successor of a particular thread. This may cause successor PEs to remain idle for many cycles. Another limitation has to do with control independence. When a misprediction occurs, all threads, starting from the misprediction point, get squashed, although many of them may be control independent of the misprediction.

In this paper we present a hierarchical technique for building threads, as well as a non-sequential scheme of assigning them to PEs, and a selective approach to squash threads in case of misprediction, in order to take advantage of control independences. This technique uses dynamic resizing, and builds threads in two steps, statically using the compiler as well as dynamically at run-time. Based on the dynamic behavior of the program, a thread can dynamically expand or shrink in size, and can span several PEs. Detailed simulation results show that our dynamic resizing based approach results in a 11.6% average increase in speedup relative to a conventional speculative multithreaded processor.

1. Introduction

As we approach billion-transistor processor chips, the need for a new architecture to make efficient use of the increased transistor budget arises. Architectures such as superscalar and VLIW use centralized resources, which prohibit scalability and hence the ability to make use of the advances in semiconductor technology [4]. Speculative multithreading (SpMT) arises as a candidate for making efficient use of the available transistor budget, because of their use of decentralized resources. There is a growing interest in the use of SpMT to speed up the execution of a single program [1] [3] [5] [7] [10][11]. The compiler or the hardware extracts threads from a sequential program, and the hardware executes multiple threads in parallel, most likely with the help of multiple processing elements (PEs) that are organized as a circular queue. Whereas a single-threaded processor can only extract parallelism from a group of adjacent instructions that fit in a dynamic scheduler, a speculative multithreaded processor can extract the coarser, thread-level parallelism (TLP).

In order to get the best performance from speculative multithreading, an efficient thread selection scheme is needed. Thread successor prediction can be performed using a context predictor [8], or any type of data value predictor that predicts the successor of the current thread to be one of the targets of the thread. However, the above model has the following limitations.

- First of all, if the thread predictor is unable to make a high-confidence prediction, all successor PEs will remain idle for many cycles.
- Furthermore, the above model cannot easily exploit control independence between multiple threads. This is because in case of successor misprediction, all threads, starting from the misprediction point are squashed, although some of them may be control independent from the misprediction point.
- Finally, the optimum size of a thread is difficult to determine at compile time. Small thread do not expose enough parallelism. On the other hand, large threads cause the following problems: (i) If a PE is assigned a large thread, it requires enormous buffering for the values generated waiting for its thread to commit. (ii)

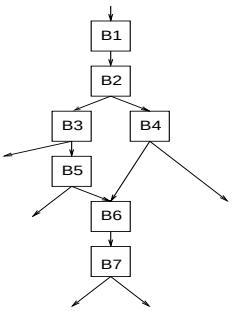

Figure 1. Control Flow Graph of a Thread

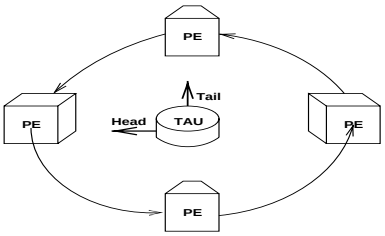

Figure 2. Circular Queue Arrangement of PEs in a SpMT Processor

This waiting time can be large if the thread is assigned to a PE that is far from the head, because threads commit in program order. (iii) Recovery actions due to thread-level misprediction can become very expensive. (iv) Memory dependence mis-speculation may increase.

An example that shows the effect of thread size on the overall performance is presented in Figure 1. The figure shows a control flow graph of a thread. Each block is a basic block. A successor predictor will have to choose a successor among the exits of B3, B4, B5 and B7. The number of unique targets can be large and undetermined at compile time. For example, a subroutine return is shown as one exit on the graph but in reality it represents several targets, as the same subroutine can be called from different parts of the code. In order to limit the number of targets to choose from, we propose predicting the successor to be one of the targets of the most probable path through the thread. In Figure 1, if the most probable path through the thread is known to be B1, B2, B4, B6, and B7, then the successor is predicted as one of the two targets of B7.

In order to get the best performance from this approach, forming the "right" threads is pivotal. For example, in Figure 1, if the most probable path is not known, then it is better to have smaller threads such as B1 and B2, B3 and B5, and B6 and B7. The traditional approach is to statically form threads at compile time [9]. Once a thread is formed, it is not modified during the execution of the program. This approach has two drawbacks. (i) Information available during thread formation may be inaccurate, resulting in "poor" threads, and (ii) the behavior of a thread may change during the execution of the program, especially if the thread size is large. Even though the first drawback can be overcome to some extent by using profile-based information during thread formation, the second drawback is a severe limitation, especially because a thread can encompass many basic blocks, and can therefore undergo substantial changes in behavior. In this paper, we use *dynamic resizing* to form threads in two steps, the first step is statically at compile time, and the second step is dynamically at run-time. Threads may be dynamically enlarged or shrunk, based on run-time characteristics.

The rest of the paper is organized as follows. Section 2 provides a brief review of the speculative multithreading architecture, as well as previous work done in building threads. Dynamic resizing scheme is presented in section 3. Our experimental setup is shown in section 4. Detailed simulation results and analysis are presented in section 5. Finally, the conclusions are presented in section 6.

2. Related Work

In this section an overview of the SpMT models is presented, together with the proposed thread selection mechanisms.

2.1. Speculative Multithreading

Most of the speculative multithreading architectures proposed so far use a circular queue organization, as indicated in Figure 2. This makes it easy to maintain sequential thread ordering. The program is partitioned into threads and multiple threads are run in parallel on multiple PEs. When there is a thread-level misprediction, all PEs from the misprediction point up to the tail PE are squashed.

Examples of prior proposals using sequential threads are the multiscalar model [3], the superthreading model [2], the trace processing model [7], and the dynamic multithreading model [1].

Pivotal issues for obtaining high performance in this microarchitecture are the thread formation, assignment, as well as squashing methods in case of misprediction.

2.2. Thread Selection

A thread is a group of connected basic blocks, and can have multiple targets. Threads may be formed at compile time by sequentially going through the static binary and adding instructions to the thread until a maximum number of instructions are reached, a call/return instruction is

reached, or the number of targets reaches a specific number[1]. The above scheme is used in multiscalar [3]. Other examples of threads formed at compile-time are: loop iterations, post-loop code and procedure calls. The thread information is conveyed to the hardware.

In some other schemes, threads are built dynamically. A dynamic thread is a single path in the control flow graph of the static thread, it is called a trace [7].

However intuitive the above schemes are, they suffer from a severe drawback, which is the size of the formed thread. If we blindly impose a size on the thread we may reach a hard-to-predict point at the exit of the thread. Furthermore, when threads are formed statically, information about the number and width of PEs may not be available. Assigning threads of inappropriate size to PEs (for example, large threads on narrow PEs) will negatively affect the performance.

In the next section, we present a hybrid method for thread selection. Threads are built using a combination of static methods and dynamic methods. Also the thread assignment mechanism and squash mechanism are modified to take advantage of the new scheme.

3. *Dynamic Resizing* Scheme

In this section we propose to use dynamic resizing — run-time analysis of code— to build threads in two steps, statically at compile time, and dynamically during the execution of a program. We also show how thread successor prediction is done using this scheme. Then we discuss the implementation details.

3.1. Main Idea

We propose building threads in a hierarchical way. The first step is done statically using the compiler. We use the technique presented in section 2.2 to build small sized threads, called *tasks*. These tasks do not change during execution. However, at run-time, tasks are concatenated together to form *supertasks*. A supertask is a group of one or more tasks. A supertask is identified by its start task, end task[2], and number of tasks. These two tasks identify the longest path through the supertask. If the start and end tasks are the same and the number of tasks is more than one, it means there is a loop inside the supertask.

Based on the number of tasks present in a supertask, we reserve the required number of PEs for it at the time of its assignment. Moreover, a supertask may shrink by pruning off some of its tasks, based on some run-time information, as will be shown later in the paper.

[1]Some of the targets may be unknown at compile time, such as subroutine return addresses. Thus the number of targets at run time can be more than expected.

[2]Control may flow out of a supertask from multiple tasks; the *end task* corresponds to the last task in the longest path through the supertask

3.2. Supertask Formation

The main concept is to form supertasks dynamically, during the execution of the program. At the start of program execution, each supertask is basically one task. The only information available is that of the static tasks formed by the compiler, as indicated in Section 2.2. Hence, no supertask prediction can be made. Therefore, at the start, only task predictions are done, and tasks are assigned to PEs in the conventional sequential manner. When a task physically commits, a check is done to see whether it is beneficial to append that task to its predecessor task(s) to form a bigger supertask. For a task to be appended to a supertask, the following conditions must be satisfied: (i) the candidate task is the most probable target of the supertask to which we are trying to append the task, (ii) the number of tasks within the supertask has not yet reached a specific threshold, and (iii) the candidate task has fewer targets than the end task of the predecessor supertask. If all of the above conditions are satisfied, then the task is appended to the corresponding supertask, and becomes its new end task; the counter indicating the number of tasks in the supertask is incremented. If the number of tasks in a supertask has reached a specific threshold, no more tasks can be appended to it. It is to be noted that a task can be appended to several supertasks. This is logical, because several supertasks may have a call to the same function.

When a supertask physically commits, that is, its end task physically commits, a check is made to see whether the end task reached is the same as the recorded one. If they are the same, a confidence counter is incremented, otherwise the confidence is decremented. Whenever the confidence reaches zero, the end task is set equal to the start task and the counter indicating the number of tasks in the supertask is restored to one. That is, the supertask is shrunk to a single task.

3.3. Supertask Prediction

Previously proposed successor predictors predict one out of N targets (where N is the maximum number of a targets for a task), by using a form of data value predictor. Performing such a prediction becomes difficult when the number of targets (and the targets themselves) keep changing dynamically.

We propose using the end task information for supertask successor prediction. The successor to a supertask is predicted to be the most probable successor of its end task. Such a prediction is done if the confidence counter is higher than a threshold. A confidence counter is a counter asso-

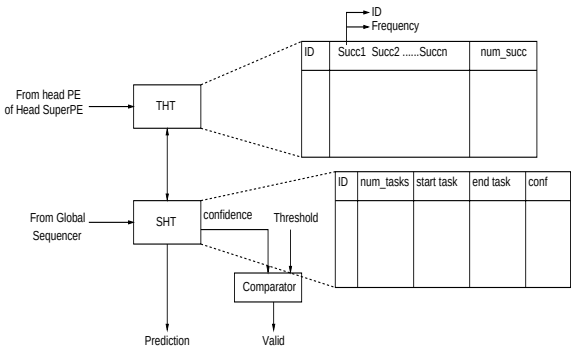

Figure 3. Block Diagram of Supertask History Table and Task History Table

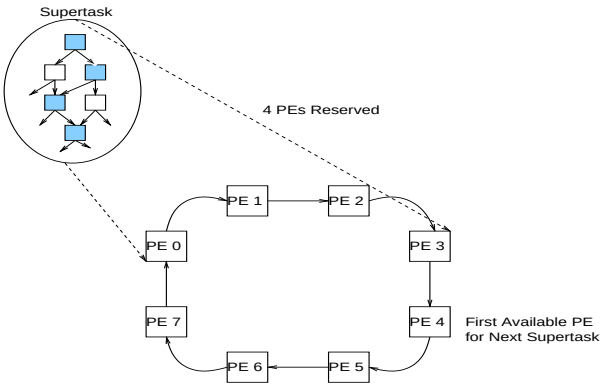

Figure 4. Supertask assignment

ciated with each supertask indicating the confidence in the correctness of the end task. Hence, supertask prediction is performed only when it is beneficial.

3.4. Implementation

The above system can be implemented using two tables: a task history table (THT) and a supertask history table (SHT). Each THT entry includes the task ID (each static task has a unique ID), the list of successors (from our simulations we found that this list does not need to have more than 10 entries) and number of successors. Each SHT entry contains the supertask ID, the start task ID, the end task ID, the number of tasks in the path from the start task to the end task, and the confidence counter. This is shown in Figure 3.

When a prediction is to be made for a successor of supertask X, the end task of supertask X is noted from the SHT to access the THT. From the selected THT entry, the successor with the highest frequency is used as the starting address of the successor supertask. The starting address uniquely identifies a supertask. Indeed we are assuming implicitly that the supertask will reach the end task, that is, no predecessor task will cause an exit from the supertask. This is because the way the supertask is formed depends on the high probability that the control will reach the end task.

Whenever a task physically commits, its successor is checked. If the successor is already in the list of successors for the task at hand, its frequency is incremented. If it is not in the list, then it is added and the number of successors is incremented. If the list of successors is already full, the successor with the lowest frequency is replaced by the new one and its frequency is initialized to one. Finally the task is checked against the end tasks in the SHT, to decide whether to add it to the supertask. At the physical commit time of a supertask, the steps indicated in Section 3.1 are performed.

Whenever a supertask is to be assigned for execution, the SHT is checked to see the number of PEs required. The number of PEs required is assumed to be the number of tasks from the start to end task of the supertask, along the most probable path. If the required number of PEs is equal to the total number of available PEs, the processor will work as a conventional SpMT processor. On the other hand, if the number of required PEs is larger than the available PEs, the supertask cannot be assigned. To avoid the above two scenarios, we restrict the number of tasks in the longest path in the supertask, as indicated above. In our experiments this limit is set at 25% of the total number of PEs. The main advantage of restricting the number of tasks is that whenever there is a misprediction inside the supertask, only PEs assigned to the supertask may get squashed. Other PEs are unaffected. Global squash takes place only when there is supertask misprediction. After reserving the required number of PEs, subsequent supertasks are assigned starting from the next available PE. This is indicated in Figure 4.

When several PEs are reserved for a particular supertask, not all the reserved PEs are assigned tasks simultaneously. First, the start task is assigned, then a task predictor makes a prediction each cycle in order to assign a task to a PE. There is a global supertask predictor that assign supertasks and reserve needed PEs, and there is a local task predictor that predicts the control flow within the supertask and fill the reserved PEs. For simplicity, we have used 4 independent task predictors. We have chosen this number because we can have at most 4 supertasks simultaneously, due to the fact that we set the threshold of the number of tasks per supertask to 25% for the available PEs. However, the predictors can be designed to use the same history tables. Thus a single update to the history tables will be felt by all the predictors simultaneously. The main purpose of having the task predictors is to check whether the most probable path is the one that is taken. If not, local predictions will establish the new path as quickly as possible.

In the next section we present the experimental evaluation of the proposed scheme.

PE Parameter	Value
Max task size	32 instructions
PE issue width	2 instructions/cycle
Task predictor	2-level predictor 1K entry, pattern size 6
L1 - Icache	16KB, 4-way set assoc., 1 cycle access latency
L1 - Dcache	128KB, 4-way set assoc., 2 cycle access latency
Functional unit latencies	Integer/Branch :- 1 cycle Mult/Divide :- 10 cycles
Number of PEs	12

Table 1. Default Parameters for the Experimental Evaluation

4. Experimental Methodology and Setup

The previous section presented a description of how tasks and supertasks are formed, assigned, and executed. In this section we present our experimental setup and in the next section we present a detailed simulation based evaluation of the proposed scheme.

Our experimental setup consists of a detailed cycle-accurate execution-driven SpMT simulator based on the MIPS-I ISA. The simulator accepts executable images of programs, and does cycle-by-cycle simulation. A post compilation step is done for the benchmarks binary, in order to demarcate tasks and convey the information to the hardware. The configuration used throughout our experiments is a processor with 12 PEs, hence, each supertask can use up to 4 PEs. Each PE is a narrow superscalar with out-of-order capability. The flow of data between the PEs is done using the same techniques used in the multiscalar [3]. Some of the hardware parameters are fixed at the default values given in Table 1. The THT and the SHT need not have more than 32K entries for all the benchmarks tested.

For benchmarks, we use a collection of 7 programs, from the SPECint95/SPECint2000 suites. The programs are compiled for a MIPS-Ultrix platform with a MIPS C (Version 3.0) compiler using the optimization flags distributed in the SPEC benchmark `makefiles`. We ran each simulation for 300 million instructions.

5. Experimental Results

In our experiments, we compare our dynamic resizing based scheme against a conventional SpMT processor (as shown in Figure 2) using a 2-level context-based predictor [8] for thread successor prediction. The level 1 table (Value History Table) of the context-based predictor has 1K entries, and the level 2 table (Pattern History Table) has 256 entries. The tables are direct mapped. Table sizes were obtained through many experiments to reach the best performance for the conventional architecture, we have tried up to 64K entries for level 2 and the results did not change much.

5.1. Speedup Obtained

Figure 5 shows the improvements in IPC using the dynamic resizing method. On average, a speedup of 11.6% is obtained. This means that dynamic resizing is able to better capture parallelism. This improvement is due in part to the multi-task assignment, that is several tasks (corresponding to different supertasks) are assigned to PEs in the same cycle, as opposed to a pure sequential assignment of tasks in conventional architecture. The second factor is the avoidance of total squash. The highest improvement is obtained for `m88ksim` due to the large amount of parallelism avail-

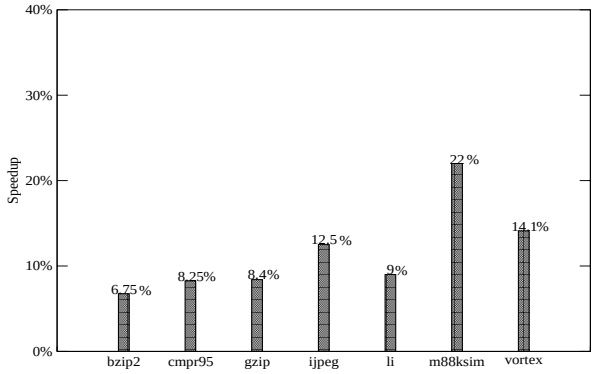

Figure 5. Speedup due to Dynamic Resizing

5.2. Statistics

Table 2 presents run-time statistics. First the number of active PEs per cycle is presented. As shown in the table, the conventional system has higher active PEs per cycle. However, as indicated in the previous section, the dynamic resizing scheme has higher performance. Hence, some of the PEs in the conventional system are doing useless work. This means the dynamic resizing is making better resource utilization, which can reflect on the number of PEs needed as well as the power consumption.

The second column of the table shows the number of distinct supertasks encountered during execution. In the conventional system a task and a supertask are the same. Supertasks are characterized by their starting PC in this statistic. It is apparent that dynamic resizing leads to substantial decrease in the number of distinct supertasks encountered during execution. This leads to smaller buffer needs for performing control prediction.

Bench	Statistics	
	Avg Active PEs/Cycle (Conven, Dyn)	Distinct Supertasks (Conven, Dyn)
bzip2	(10.33, 9.78)	(233, 114)
cmpr95	(10.51, 7.21)	(251, 149)
gzip	(10.66, 9.15)	(470, 318)
ijpeg	(8.66, 7.65)	(604, 292)
li	(7.15, 4.76)	(779, 526)
m88	(10.01, 9.07)	(693, 430)
vortex	(8.73, 7.52)	(4345, 2044)

Table 2. Run-time Statistics

Finally, Figure 6 shows the task and supertask prediction accuracies. It is to be noted that they are related. This is because a supertask prediction is done based on the prediction of the end task successor of a supertask.

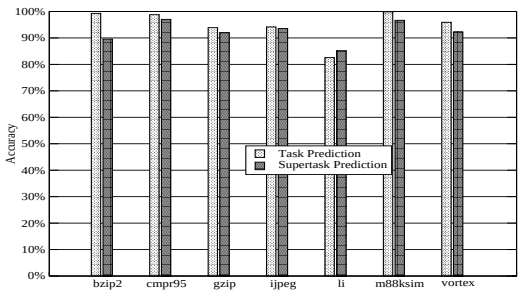

Figure 6. Supertask and Task Successor Prediction Accuracy.

6. Conclusions

In this paper we have proposed *dynamic resizing*, a new method for building threads in an hierarchical way. The first step is static, is done by the compiler, and generates tasks. Tasks are concatenated dynamically to form supertasks, which are the threads assigned to PEs. In this new method, a thread can span several PEs.

Dynamic resizing achieves an average speedup of 11.6% over the conventional architecture. Furthermore, the proposed technique makes better use of resources, this can reflect positively on the power requirement and the PEs needed. Moreover, a smaller number of distinct supertasks are obtained as compared to the conventional system, which leads to smaller storage requirements for performing control prediction.

Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) (through grants CCR 0073582 and CCR 9988256) and Intel Corporation.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proc. 31st Int'l Symposium on Microarchitecture*, 1998.

[2] J-Y. Tsai et.al. Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading. *Journal of Information Science and Engineering*, 14, March 1998.

[3] M. Franklin. *Multiscalar Processors*. Kluwer Academic Publishers, 2002.

[4] K.Olukotun, B.A.Nayfeh, L.Hammond, K. Wilson, and K.Chang. The case for a single-chip multiprocessor. In *Proc. 7th international conference on Architectual support for programming languages and operating systems(ASPLOS)*, 1996.

[5] P. Marcuello and A. Gonzalez. Clustered speculative multithreaded processors. In *Proc. Int'l Conference on Supercomputing*, pages 20–25, 1999.

[6] I. Martel, D. Ortega, E. Ayguade, and M. Valero. Increasing effective IPC by exploiting distant parallelism. In *Proc. Int'l conf. on Supercomputing*, pages 348–355, 1999.

[7] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proc. 30th Annual Symposium on Microarchitecture (Micro-30)*, pages 24–34, 1997.

[8] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical report, University of Wisconsin-Madison, 1997.

[9] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proc. International Symposium on Microarchitecture*, pages 81–92, 1998.

[10] M. Zahran and M. Franklin. Hierarchical multithreading for exploiting parallelism at multiple granularities. In *Proc. 5th Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-5)*, pages 35–42, 2001.

[11] M. Zahran and M. Franklin. A feasibility study of hierarchical multithreading. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2002.