Adaptive Partitioning Schemes in Multi-threading Processors

Introduction

In this project, we try to propose an adaptive scheme to find optimal partitioning policies in multi-threading processors. In multi-threading execution environment, sequential program is partitioned into parallel threads. These threads are executed by multiple thread units. The partitioning process can be implemented either by compiler or by some run-time hardware structures. The partitioning policy directly affects the level of thread-level parallelism that can be effectively extracted, thus, it has a critical impact to the performance of multi-threading processors.

Existing thread partitioning policies include fix-size trace partition, mem-slice partition, loop partition, and procedure partition. The features of these policies are summarized in table 1.

We have made our study into two parts. In the section 1, we first propose a multi-threading microarchitecture that incorporates parallel execution model of multiple threads, control speculation, data speculation and multi-threading execution environment. Any existing partition policy can be applied to this architecture. Some related compiler techniques are also discussed. We extended this architecture by introducing some adaptive partition schemes that can evaluate the performance of past dynamic threads and intelligently select the optimal partitioning policy at run-time. We also discuss some important issues and steps toward future study on this topic.

In section 2, we evaluate some existing partitioning policies and some of their variations by simulating these policies on a simple multi-threading processor model. We expect to collect some insightful knowledge on how these policies perform on different programs. Our preliminary experimentation can serve as the starting point to further this study with more detailed microarchitecture designs.
<table>
<thead>
<tr>
<th>Method</th>
<th>Features</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>• Fixed size instruction chuck</td>
<td>• Good load balance</td>
<td>• No knowledge of dependencies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Can select bad partitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Need to keep history</td>
</tr>
<tr>
<td>MEM-Slice</td>
<td>• Partition at memory references after skipping minimum thread size</td>
<td>• Good load balance</td>
<td>• Create threads randomly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• May eliminate one level dependencies</td>
<td>• No knowledge of dependencies</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Can select bad partitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Difficult to keep history</td>
</tr>
<tr>
<td>Loop</td>
<td>• Natural loop boundary as thread partition</td>
<td>• Low creation overhead</td>
<td>• High inter-thread dependencies</td>
</tr>
<tr>
<td></td>
<td>• Innermost loops only</td>
<td>• Speedup do-all loops</td>
<td>• Parallel only for innermost loops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Simple fetch engine to all TU</td>
<td>• TU idle at sequential exec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Squash at outer loops</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Squash at control misspeculation</td>
</tr>
<tr>
<td>Procedure</td>
<td>• Leaf procedure threads</td>
<td>• Low creation overhead</td>
<td>• Bad load balance</td>
</tr>
<tr>
<td></td>
<td>• Non-leaf procedure threads</td>
<td>• Good control speculation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• After procedure threads</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Existing partitioning policies

Part 1. The adaptive partitioning multi-threading microarchitecture

We first describe a general multi-threading microarchitecture without adaptive-partition schemes. All the related issues and techniques will be discussed. At last, we introduce how the adaptive schemes can work on our proposed architecture.

Machine Model

The proposed microarchitecture is shown in Fig. 1.1.

The multi-threading processing unit

The multi-threading processing unit consists of a central thread arbiter, a global/shared register file, a global/shared L2 data cache and multiple thread units. The N thread units
are organized in a circular ring structure. Each thread unit has local register file, local load/store queue and a validation unit. The inter-thread data dependent values can be forwarded to other thread unit by the data forward network. For in-order thread execution model (fixed size, mem slice, loop, and etc.) the data forward is unidirectional, resulting in the unidirectional circular ring structure. While in the out-order thread execution model (procedure), the data forward can be bi-directional. Among the N parallel threads, only one thread can be non-speculative thread. This thread stays in the head of the circular queue (in-order model) or the root node of thread tree (out-order model). The rest up to N-1 threads are speculative ones. The validation unit validates whether a thread execution is successful or not at the time when the head thread unit finishes all its instructions. The successful thread execution means both the run-time thread and all the corresponding data values are correctly predicted. If a thread is validated, all local register values and stored values are committed to global machine, and some information will be updated in different tables. Otherwise, all threads will be squashed. The thread arbiter fetches instruction stream from I cache or trace cache, then dispatches speculative threads to thread units. It monitors all thread units and coordinates thread execution by sending commands to thread units. The commands include “create thread”, “validate thread”, “commit thread” and “squash thread”, and etc.

Fig. 1.1. Overview of the multi-threading architecture
Thread partition unit

The thread partition unit breaks sequential program into parallel threads based on partition policies. The instruction stream of a thread is put into trace cache upon partition is done and the entry in thread table is created. Various partition policies can be implemented in thread partition unit. The adaptive scheme also takes effect in this unit. We will discuss the adaptive scheme later.

Branch predictor

The branch predictor can be either traditional centralized structure, or it could be distributed to each thread unit. The output of the multiple branch predictors gives a bit vector that can be used to identify next speculative thread. In order to facilitate the multi-threading execution environment, the branch predictor must be able to operate on speculative thread and avoid misspeculated results affecting the “clean” branch history. Chansik may give some insights on how branch predictor works on multi-threading environment.

Trace cache and I cache

Trace cache keeps dynamic instruction stream of a thread in a cache line. We derive the existing trace cache design to our architecture. There may be extra pointers or indexes that link a trace cache line to corresponding entries in other structures. Other than that, there is not much difference from the trace cache structure in our model to the existing design.

The instruction cache is also the same as the traditional structure.

Thread Table

Each entry of the thread table keeps information of a run-time thread. The table can be organized as a single entry list or a set-associative table. The entry/set can be indexed by partial thread starting address or a hashed number of different thread attributes. The entry consists of following fields:

- Thread ID: each thread has an ID. It could be the index of the table or other ID number.
- Thread staring address, ending address: PC of the first and last instruction of the thread.
- Size: number of instructions in thread.
- Branch flag: number of branches in thread, whether the last instruction is a branch.
- Branch history: bit vector of branch instructions.
- Target address, fall-through address: when the last instruction is a branch instruction, target address keeps the next instruction address when the branch is taken, fall-through address for branch untaken.
- Pointer to trace cache: the pointer to the corresponding entry in trace cache, that keeps the run-time instruction stream of the thread. Trace cache and thread table can be indexed by same ID.
- Pointer to value prediction table: a pointer to the corresponding entry in value prediction table. The entry is called register value pattern table, which predicts register values on thread level (see description of value prediction later). It can be indexed by thread ID; or two tables can be combined together.
- A list of possible succeeding threads: this list is used to predict the next thread. Based on thread execution history, the succeeding threads of a given thread can be recorded in a list, with some run-time attributes, e.g., frequency of appearance, and a context flag. We use the context flag to record a snap shot of a given thread execution with the context of other parallel threads that have been executing simultaneously with it. The flag hashes all parallel thread IDs and/or branch flags so that the run-time context is recorded in this flag. (Fig. 1.2.) There are different ways to implement the list. For example, the successor of a given thread can only start from two addresses: target address and fall-through address. If threads starting from the same address can be organized in the same set in set-associative thread table, all the possible successors can be retrieved by looking up two lines that are indexed by target address and fall-through address. In this approach, there is no need to maintain a separated list in the thread table entry; the context flag and other run-time attributes can be combined with some other fields in thread table entry.

![Fig. 1.2. Context flag of a thread](image-url)
Thread prediction table

This table is used to predict next speculative thread based on thread execution history. Similar to branch prediction techniques, we use M run-time threads to predict the next thread. These M threads include N currently executing threads, and M-N successfully committed threads. M should be larger than N, because, among the N currently running threads, only one thread (on the head thread unit) is non-speculative, and even this non-speculative thread may not be able to successfully completed. If M is not large enough, using too many speculative threads to predict may not give good accuracy.

The thread prediction table predicts next thread based on previous M committed threads. The table can be indexed by a hash function of M thread IDs, e.g., the XOR of all IDs. Each entry can either have a specific thread ID to indicate the predicted thread ID, or, each entry can maintain a list of candidate threads with ID and a context flag.

When a thread prediction table entry is hit with single candidates, this thread will be selected as the predicted thread. When multiple candidates are available, various techniques can be used.

First, the thread table entry of the newest thread (the last one in previous M threads) will be looked up. Remember that thread table entry also contain a list of possible successors with context flags. The thread IDs with matching flag will be selected. Then, these IDs will be used to locate corresponding thread table entries.

Second, branch predictor can be looked up for K branch predictions, where K is the multiple branch predictor throughputs. The result will give a K-bit vector. This bit vector can be used to find a match in the candidates by checking the branch history vector of each thread.
Third, a counter can be maintained in the thread prediction table entry. The specific scheme is not quite clear before some analysis to control flow behavior of run-time threads. And it is related to specific implementation. For example, if the table entry only keeps two most frequent candidates, an up-down counter can be used to select from two candidates.

If no match can be found, either partial match or some heuristics can be used to find the best match. The structure of thread prediction table is illustrated in Fig. 1.3.

**Value prediction table**

The tight dependency chain in sequential program tends to results in inter-thread data dependencies that will serialize the execution of parallel thread. We propose a mechanism to target live-in data value prediction of parallel threads. The inter-thread data dependencies take place at live-in values of each thread. Intuitively, the value prediction accuracy is related to the level of inter-thread dependencies, or in another words, the more live-in values a thread has, the less effective the value prediction is likely to be. We only focus on register value prediction. The live-in memory data value takes place at load instructions. Such situation can be handled by existing load value prediction method.

The live values can be classified as unpredictable values, predictable values, and constants. The unpredictable values may include values loaded from memory (this can be handled by load value prediction), results calculated from two register operands (in this case the value may be difficult to predict because of the two operands), or some FP values. The predictable values have more regular behaviors, for example, results calculated from single register operands (move register or single register operates with an immediate), the loop index values and induction values also fall into this category. The constants are values that will not change in a relatively long period.

The entry of value prediction table is referred to as register value pattern table. Each thread has a register value pattern table that gives the information of the changes to each register value during the execution of this thread. The thread table entry may have a pointer to the corresponding register value pattern table; or the register value pattern table can be embedded in thread table entry.

In register value pattern table, the behavioral patterns of each register can be marked by live-in and live-out attributes.

- Live-in, unpredictable: live-in value is not predictable.
- Live-in, predictable: live-in value is predictable from previous threads.
- Live-in, constant: live-in value is a constant that can be read from global register file.
- Live-out, load: the last update to this value is loaded from memory by this thread.
• Live-out, unpredictable: register is updated by this thread. Value cannot be predicted.
• Live-out, predictable, reset value: the register is updated in this thread, the value from last update can be predicted if it is reset to a new value, the value is specified.
• Live-out, predictable, stride: if the new value is based on its initial value (it is also a live-in value of this thread), a stride is given as the change made by this thread to the initial value. The predicted live-out value is initial value + stride.

The live-in and live-out attributes can be combined. For example, a register can host both predictable live-in value and unpredictable live-out value. A bit pattern can be used to incorporate these attributes.

The stride prediction model is simple. The live-out values are always represented by a stride value (constant) plus an initial live-in value. This simple model may not be good enough to cover all conditions where the register values are changed. For example, in some loop iterations, a register value can be shifted 1 bit each time. If the loop iteration is considered a thread, the stride model cannot predict the shifted value since the stride value is changed every time. The stride model has only a constant value as the parameter. We can extend this model by adding an operation code, e.g., add/sub, mul/div/shift, and etc. Now each value prediction entry has two parameters, OP and constant. The predicted live-out value can be represented as initial live-in value OP constant. This will give a better prediction accuracy compared with the simple stride model.

We predicted register values in thread level. Each register value pattern table can stay as an entry in a global value prediction table that is associated with an entry in thread table; or, the table can be a field of a thread table entry.

When a speculative thread is predicted as dispatched to an idle thread unit, its live-in values are predicted. Its register value pattern table is looked up and matched to the existing parallel threads that are predecessors of the new thread and produce live values to this new thread. For any given live-in registers of the new thread, if a match is found that this register value is also the live-out value of its closest predecessor, a decision is made to predict this live-in value based on corresponding attributes. For example, suppose that register R holds a predictable live-in value of thread M and R3 also stores the live-out value of thread M-1, the following conditions can take place.

1. If R is marked as a constant, its value is read from global register file.
2. If R has a predictable live-out value of thread M-1 with an OP and a stride value, the live-in value of R in thread M will be set as (initial value of R in thread M-1) OP (stride value of R in thread M-1).
3. If R is a live-out value of thread M-1 with a specified live-out value in the register value pattern table of thread M-1, this value is given to thread M as the initial value of R.
4. If R is a load value in thread M-1, the load value predictor is used to predict the live-in value of thread M.
(5) If \( R \) is not predictable, do not predict its live-in value in thread \( M \). Synchronization has to be used.

If a live-in value of a thread is marked as unpredictable but the communicating live-out value in the preceding thread is marked as predictable, the decision on whether to use prediction is put into a question. There are two approaches. In the aggressive method, whenever the live-out values are predictable, the predicted values are always used to start the execution of the new thread. In the conservative solution, the prediction is made only when both live-in value and live-out value are marked as predictable. A confidence counter can be used to guard the predictable/unpredictable bit. If the last prediction is successful, the counter is incremented; otherwise decremented.

During the commitment stage of a thread, the thread is validated and the results of its operations will update the global machine. Its live-out values are also compared with the initial live-in values in the existing threads that are its successors. If any mismatch is found, the corresponding threads will be either completely squashed or partially restarted; and all related register value pattern tables are updated with the information on whether the prediction is successful.

The structure of value prediction table is shown in Fig. 1.4. A special set of data – initial register values should reside in each thread unit. The initial register values come from either global register file or value predictor. This is used to validate value prediction and restart operations using mispredicted values.

**Fig. 1.4. Value prediction**
The operation model

Based on the structures and their mechanisms explained above, our proposed multi-threading processor (without adaptive partitioning scheme) can work as follows.

(1) At program startup, there is no enough run-time knowledge in branch predictor, trace cache, thread table, thread prediction table and value prediction table. Prediction is not available. Therefore, some predefined scheme, e.g., fixed size, can be set in the partition unit to partition the program into thread. At this time the threads may not be executed in parallel. During this period, all the tables are filled with run-time information. The thread table entry is filled with each thread from partition unit. The live-values of each thread are analyzed and their behaviors are transferred into some pattern bits in the register value pattern tables. In thread prediction table, a hash function gives an index based on previously committed M threads, and then the corresponding table entry is filled with the next thread ID.

(2) After certain period of execution, the information in all tables is enough for prediction. For example, based on previously committed M threads, an index is calculated by hashing the M thread IDs. If the corresponding entry in thread prediction table is not empty, this is, the next thread is "predicted" at the moment, trace cache is looked up for this thread. If trace cache is hit, the instruction sequence will be delivered to an idle TU. The TU will start the thread as a speculative thread. At this point, the value prediction table should give some meaningful information since the past run-time history is repeated. The initial register values of the new thread are either copied from global register file or predicted based on information in register value pattern tables. Regarding the unpredictable values, since the dependency information (where the live value comes from) is known, synchronization can be used to coordinate thread execution. The new thread then starts to execute.

(3) If no prediction is available at a certain time, that is, a new section of program is executed without past run-time history, no speculative threads will be created. After all existing threads are committed or squashed, the sequential execution is started while the partition unit keeps breaking the program into threads. This condition is similar to (1), where no history information is available.

(4) Speculative thread can be used to predict next speculative thread and its initial register values, as described in (2).

(5) If there are multiple candidates available in the thread prediction table, various techniques can be applied to select the most likely thread, as discussed previously.

(6) The non-speculative thread is validated at the commitment stage. The committed threads will update all tables – thread table, thread prediction table and value prediction table. Whenever a thread is proven to be wrong, this thread, as well as all the succeeding
speculative threads will be squashed. All these wrong speculative threads will not update the tables. But the wrong non-speculative thread will add information to the tables as the first unsuccessful thread after a series of successful one, because its predecessor must be a committed thread. That is, the thread misspeculated by correct threads will be recorded. The further misspeculated thread predicted from threads that are already misspeculated will not have a history.

**Load balance and thread overhead**

We have discussed control speculation and data speculation techniques in multi-threading processors. In order to achieve better efficiency of multiple thread units, load balance and thread overhead must be considered.

Load imbalance greatly affects the efficiency of parallel thread units. Load imbalance takes place in two conditions.

1. Not enough parallel threads are available to feed all thread units. This happens at program startup, when no history information is available to make prediction. Also, according to some specific partition schemes, e.g., procedure, the number of simultaneous threads can be very low at a give time.

2. The simultaneous threads in execution differ greatly in size, so smaller threads spend a lot of time on waiting for previous larger threads to commit.

The loop and procedure schemes suffer from load imbalance due to both (1) and (2). Fixed size of mem slice schemes work well because all threads are in similar size.

During program startup, there is no knowledge to feed all thread units with simultaneous threads. One method is to use compiler or profiling information. The compiler or profiler can preset the thread table, thread prediction table, value prediction table, as well as the trace cache, by static analysis or profiling information. Therefore, at any given time of execution, there are always enough parallel threads to feed all thread units.

To improve load balance, the partition unit should keep all simultaneous threads in balanced size. When a thread is predicted but its size is not either too big or to small compared with existing thread in execution, the partition unit may have to change its decision to break the thread in a new instruction to make more balance load. This is related to adaptive partition scheme discussed later.

Thread start and retire overhead is closely related to detailed implementation. In existing schemes, loop and procedure have low thread overhead, while the overheads in fixed size and mem slice are costly. This is because random threads in fixed size and mem slice are likely to cause some conflict in the thread table or prediction table, or it takes long time to look up large tables that contains all random threads; while the number of loops and procedures is not that large. Also, random threads are not likely to have different control paths, thus it may take long at the retirement stage to validate whether the speculative path is correct. Loops are easy to predict and likely to follow the same path.
We certainly want to find some regular behaviors that can avoid random partition that will create too many threads. One way is to find some “anchor points” as the thread boundary. Thread partition is only allowed at anchor instructions. In this way the number of thread will be reduced and the threads will be of more regularity. The detailed analysis should be performed by experiment.

**Compiler support**

Various compiler techniques can be helpful to multi-threading execution.

1. Compiler marks synchronization code for live values. This is implemented in Multiscalar processors.
2. Compiler generates live values early in the threads and use local copies to intra-thread usage. The early live-out values can be forwarded to other thread units to resolve inter-thread dependencies, so that the dependent threads will not be stalled on synchronization. This technique is used by Illinois Superthreading and Stanford Hydra to deal with loop-carried dependencies.
3. Compiler marks the life range of specific values. The live range starts at a write and is followed by a series of reads. If compiler can generates some code to specify the life range of a value, e.g., how many reads are following before next write, mark the last read to specify the termination of a life range, and etc., the run-time partition scheme can use such information to decide whether to partition a new thread within a certain value's life range (will cause inter-thread RAW dependency), or not to create new thread until a certain life range is completed. I have noticed some cases when the life range of control registers are separated by thread boundary. The control registers are HI, LO, and FPFLAG. For example, if one FP instruction will affect the FPFLAG, and a later branch instruction will read the FPFLAG and make decision, these two instructions should stay in one thread. Otherwise many wasted cycles will be spent on synchronization. This kind of unnecessary cost can be avoided by compiler analysis on life range.
4. Compiler marks register spill/refill. In many cases when register resource is limited, some values will be put back to memory and reloaded later to registers. This kind of behavior will cause the RAW dependencies on memory locations that are not really necessary. Compiler doesn't know the run-time thread boundary. But it knows for sure the cases of spill/refill, which are different from regular memory accesses. If compiler can mark the spill/refill to distinguish the cases from normal load/store, such information may be helpful to make run-time decisions. For example, if there is a register live-in value that is refilled from memory, we can bypass this instruction that requires resolving RAW dependencies on that memory location. The value can be forwarded directly from the previous thread where the spill code resides in. If the value is not changed in the previous thread where spill takes place (this may be common in the case of spill/refill), even the register dependency may not exist.
5. Similar situations are saving/restoring values on procedure calls. All these unnecessary dependencies may cause some threads to be stalled. If compiler can mark such conditions, these dependencies will not even take place.

6. To avoid the sequential execution on program startup, compiler can pre-partition program by static analysis or profiling can be used to provide some priori knowledge before program execution. At the compiling/profiling time, some history information can be stored in related tables. During program startup, the tables can be filled with static/profiling information so that prediction is available to create enough parallel threads.

Adaptive scheme

We extend our machine model by incorporating adaptive partition schemes based on existing structures.

According the past thread execution history, a run-time decision can be made to alter the partition policy by the partition unit. We propose this adaptive decision making logic by extending more history information in thread table.

As explained previously, each thread table entry has a list of candidate threads as its direct successor. We add following information to the list:

- Number of successful executions, this tells how many runs have been successfully executed.
- Number of unsuccessful executions and the reasons why they fail, e.g., control miss, data miss.
- A score given by a combination of run-time statistics telling how “good” the run-time performance is. The statistics include, useful computation time, thread overhead, synchronization time, idle time, waiting for commitment time, and etc. The score can be represented as the rate of useful computation time over total execution time. This gives how efficiently the thread unit has been utilized on past successful executions.
- Any other information can be used the measure the performance of a run-time thread can also be kept in the list. For example, dependency information can be used to facilitate this decision-making logic. As mentioned previously, compiler can place some special marks on life-range and dependency chains. If a thread boundary is across the short life-range of some unpredictable values, either data misprediction or synchronization may have taken place to slow down the succeeding threads.

Given the above information, the partition logic can monitor the simultaneous threads can evaluate their values at run-time. If any thread is found to be a “bad” one, this thread will be discarded and an alternative partition schemes is applied to it. Such intelligent run-time decision-making logic is referred to as “adaptive partition scheme”.

We propose two different approaches to achieve adaptive scheme and their combination.

Forward adaptation

In the forward adaptive scheme, the partition unit monitors the successors of newest thread. If this thread has very few instances of successful successors, or among the successful successors, very few of them have “good” performance rating; this implies that this thread may be a “bad” one. For example, if the successors of a given thread have poor performance, the reasons for such performance should be recorded, such as, synchronization, misprediction, load imbalance, breaking the short life range of some unpredictable values, and etc. The decision-making logic should attack these reasons that have resulted in bad performance. A decision can be made to either shrink or grow the thread size to balance the load or fit the register life rage. Or, the partition scheme can be altered, e.g., change fixed size to mem-slicing. The history of this new thread will be stored if it can be completed successfully.

This scheme looks the successors of a given thread to decide whether this given thread should exist. This is so called "forward adaptive" scheme. See Fig. 1.5.

Backward adaptation

The backward adaptive scheme looks at the predecessor of a predicted thread. That is, when a thread is predicted, the history of this predicted thread is examined. If its history is "bad", this implies that its predecessor may not be a "good" thread. Therefore, a
decision may be made to break the predecessor thread at the different place. The compiler analysis also applies to make the decision, as mentioned earlier. Backward adaptation is illustrated in Fig. 1.6.

Both forward and backward adaptive scheme examine the next predicted threads and alters the partition policy on the newest (most speculative) thread. The difference is, the forward adaptation checks all successors of the newest thread, if none of them is good, the newest thread is bad for sure and it should not be executed any more. While the backward adaptation only examines the predicted successor of the newest thread, if the predicted thread is bad, the newest thread should be squashed only this time. It does not necessarily mean that the newest thread is always bad; it could be a good one in other contexts. Also, the forward adaptation only looks one step forward. The backward scheme can look more steps back, that is, the run-time decision can also squash more predecessor threads rather than the most speculative one. Forward adaptation is performed before thread prediction; backward schemes takes place after next thread is predicted.

**Bi-directional adaptation**

The "forward" and "backward" schemes can be applied together, that will lead to a more complicated adaptive decision-making logic. To expedite the evaluation of whether a thread has “good” successors, an additional rating of measure how “good” its successors are can be attached to each thread. If a thread has a poor rating on “good” successors, it might be discarded; and its predecessors will also be downgraded on their ratings because they have a “bad” successor. In this way, the evaluation can be performed either forward
or backward with more flexibility; but the logic becomes more complicated and difficult to handle. Many heuristics can be applied on them. The effectiveness cannot be evaluated until detailed simulation is performed.

**Suggested future work**

The future study on this topic can be outlined to the following schedules.

1. Analyze the control flow behavior in thread level. Implement an effective thread prediction mechanism that gives high accuracy prediction. The high accuracy is very important in the multi-threading model. At each time there is only one non-speculative thread, while the rest N-1 threads are speculative based on prediction. These N threads (or more) are used to predict the next thread with N-1 predicted threads. We should take all N parallel threads as the target of prediction (it's similar to the concept of working-set). The high accuracy for a single thread prediction is not enough. The different partitioning schemes should not bring significant effect to the accuracy of this thread predictor.

2. Based on 1, all live values are analyzed and categorized by their behaviors to unpredictable, predictable and any other distinctive behaviors. A register value pattern table should be attached to each thread. Upon the thread is created on a TU, this table is copied to the local register file of the TU and some information will be used to determine detailed operations on value prediction or synchronization. Different schemes may change the result of this value analysis notably.

3. Based on 1, 2, and the proposed multi-threading architecture, the compiler can perform some static analysis regarding issues mentioned previously.

4. Implement the proposed architecture and do simulation. The current architecture can only count cycles. There should be real function units that update local/global registers, local/global cache, global thread predictor, local/global value predictors, and some structures to make validation, commitment and etc. The simulations for existing schemes can be performed on this more detailed machine model.

5. Evaluate the partitioning schemes at run-time and make adaptive choices. What the key factors to determine whether a scheme is "good" or "bad" is not known at this moment. The detailed implementation of forward, backward and bi-directional adaptation, as we proposed previously, is also not quite clear so far. It should be based on extensive simulations and analysis to results.
Part 2. Simulation

We use a simple execution model to simulate multi-threading execution environment. Five existing partition policies are implemented and simulated respectively. In this section, we introduce the machine model we use to perform the simulation, analyze the results and point out the limitation of current simulation model and suggestions for future studies.

Experiment model

We use a trace-driven execution model to measure the performance of a simple multi-threading processor when applying different partitioning policies.

![Trace-driven execution model](image)

The trace producer uses the thread partitioning policies to break sequential program into traces and puts all traces into the trace buffer. The traces in the buffer will be consumed by the trace consumer that incorporates our multi-threading machine model. A cycle-by-cycle simulation is performed on the trace consumer to measure the impact of different partitioning policies on the performance of the multi-threading processor. We give more details of these structures in following text.

Trace Producer

The overview of the trace producer is illustrated in Fig. 2.2.
Instruction stream is derived from the SimpleScalar simulator. The partition unit parses the instruction stream while employing different partitioning policies to break sequential program into traces. All accesses to register and memory from a given trace are recorded into data access table. Up to recent N traces are recorded such information, where N is the number of thread units in our multi-threading machine model. When a new trace is partitioned by the partition unit, a series of operation is performed on different structures. First, the trace table is updated with the information of this new trace. The organization of trace table is similar to trace cache. Each dynamic trace is assigned an entry in the table that records program counter, branch history, live-in and live-out values, and etc. Second, the entry of the newest trace in data access table is completed by checking inter-trace dependencies to its N–1 predecessors. The table is organized in the circular queue. Each time a new trace is to be created (right after the current trace is completed), the oldest (Nth) table entry is cleared for the next trace. Finally, the instruction steam of current trace combined with detailed dependency information is kept to an instruction buffer; a new entry for the current trace is allocated in the trace buffer as well. The structure of these buffers will be introduced in next section.

We employ different partitioning policies in the partition unit. This is the only structure that different policies will affect in the whole trace producer part. The flow chart is shown in Fig. 2.3. Only the decision making part will involve the partitioning policies. Rather than that, all the rest structures will remain intact when various polices are applied.

Trace buffer

Fig. 2.4 gives the overview of this intermediate structure between the trace producer and consumer. There are two buffer structures. The instruction buffer extends instruction format with dependency information. Each instruction has two extra fields that are pointers to previous instructions where true dependencies, if any, are coming from. The multi-threading machine model will use this information to synchronize for true dependencies. Each entry in the trace buffer is designated to a particular trace with its sequential ID, the pointer to its instruction stream in the instruction buffer and its corresponding entry in the trace table. Both buffers are organized in circular structure.
Trace producer puts new entries to the tail while trace consumer takes entries from the queue head.
Trace Consumer

The structure of trace consumer is illustrated in Fig. 2.5. We incorporate a simple N-unit multi-threading processor model to consume the traces generated by the producer part. Each thread unit (TU) has its local register file that serves as the local register renaming unit, local load/store queue that hold all memory operations before updating global memory. All N TUs are organized in a circular ring. This implies the in-order multi-threading execution model. All threads will start and complete in program order. The thread unit arbiter monitors all thread units. When there are idle TUs and parallel traces are available, the arbiter takes the traces from the trace buffer and distribute them to available thread units, and send commands to start thread execution accordingly. When the arbiter sees the head TU (the oldest one) is ready to complete its trace, it sends command to the head unit to commit the thread.

The operation of each thread unit is defined as a finite-state machine. The state transition logic for each thread unit is shown in Fig. 2.6. There are seven states in this highly abstracted machine model.

- **Idle:** no thread is assigned. TU is waiting for the command from arbiter.
- **Thread creation:** TU received “create thread” command from arbiter. The whole trace body with dependency information (pointers to instruction buffer) is passed from the arbiter. The global register file is copied to local. A fixed P cycles are spent before state transition is made to next state. P is referred to as the thread creation overhead. The value of P can be set in our simulator.
- **Pipeline startup:** we did not simulate detailed pipeline structure. For simplicity, we assume each class of instruction will spend certain cycles on execution. Therefore, it is necessary to take pipeline cold start into account by adding this conceptual stage into the machine model. The pipeline startup can lead to a Q-cycle overhead. The value of Q can be specified as well.
- **Computation:** after pipeline startup cycles, we employ out-of-order and multiple issue model to thread unit execution at computation stage. The logic of computation stage will be elaborated later.
- **Retirement:** when all instructions of the current trace are completed, computation state will transfer to retirement stage. In this stage, the local register file and memory queue are buffered into some intermediate structures between local thread unit and global machine. Initially we only have one final commitment stage where a thread unit can update the global machine. We find out that threads spend a lot of time on waiting for becoming the head unit. To reduce this overhead, we pipeline the thread commitment stage into (early) retirement and (final) commitment stages. Thread unit can perform early retirement in parallel, while final commitment must be serialized to update the global machine. We assume a thread unit needs R cycles on early retirement stage. After this R cycles, if the thread unit has already been the head, final commitment will start. Otherwise, the thread unit keeps waiting until it becomes the head unit.
• Waiting for commitment: after early retirement is completed, a thread unit keeps waiting until the arbiter sends command to designate it to become the head unit.

• Commitment: this stage only takes place on head unit. The arbiter monitors all TUs every cycle. When the head TU is in retirement stage or waiting for commitment stage, the arbiter sends the command “commit thread” to force this stage transition. The global machine is updated by buffered data in retirement stage. We suppose an S-cycle thread commitment overhead. After the head TU commits, the state transfers to idle state to wait for new threads as it is put to the tail by the arbiter. The arbiter then selects next TU as the new head TU.

![State transition graph of thread unit]

Fig. 2.6. State transition graph of thread unit

We incorporate multiple-issue and out-of-order execution model on each thread unit. The computation stage of a thread unit consists the operations of M parallel function units. Each function unit is also defined as a finite state machine. Therefore, the computation stage has M parallel finite stage machines.

When the computation stage is entered, each function unit is trying to get a ready instruction from instruction window. If a ready instruction is available, the FU will remain busy for K cycles on executing this instruction. Otherwise, the FU will wait one cycle and look for a ready instruction again. The M function units run their own finite stage machine respectively. Whenever the instruction window is empty, the window is filled with new instructions remaining in the thread. An FU goes idle when no instruction is available in the trace, that is, all instructions are either completed or being executed on other FU. After all function units become idle, the current thread is completed. The
computation stage is finished. The state transitions to retirement. The finite stage transition graph in computation stage is shown in Fig. 2.7.

The thread unit can be defined by issue bandwidth and instruction window size. The parameters can be set to different values in our simulator. We can alter these parameters to measure the performance of different machine configurations. Especially, when instruction window size = 1, the out-of-order model becomes in-order model. Similarly, when issue bandwidth = 1, the single-issue model is defined.

**Simulation results**

Based on the above producer-consumer trace-driven model, we simulated multi-threading execution on different partitioning policies. We performed a cycle-by-cycle simulation on our proposed multi-threading machine model.
We assume the multi-threading processor has eight thread units. Each thread unit runs its own FSM during the whole simulation process. Each stage has a series of counters to accumulate cycles that are spent on each state or sub-state. These counters increment in units of issue slots. This allows us to break down execution time to different make-ups on each state.

The thread unit is defined with different configurations. For simplicity, we simulated four machine configurations.

- Single issue, in-order: issue bandwidth = 1, instruction window size = 1
- Single issue, out-of-order: issue bandwidth = 1, instruction window size = 8
- Dual issue, in-order: issue bandwidth = 2, instruction window size = 1
- Dual issue, out-of-order: issue bandwidth = 2, instruction window size = 8

We also made following assumptions on thread overhead.
- Thread creation: 5 cycles
- Pipeline startup: 4 cycles
- Early retirement: 3 cycles
- Final commitment: 2 cycles

We use simple criteria to classify instructions into different delay models, such as, 1 cycle for INT ALU, 2 cycles for load/store, 4 cycles for FP ALU.

We think above assumptions are fair to all partitioning schemes. We assume a perfect-synchronization model. Only unresolved true dependencies require synchronization. The thread creation, pipeline startup and thread completion overhead can be measured by setting the values of P, Q and R in our simulator. Since our machine model only works for in-order multi-threading execution, we did not simulate procedure partition policy that requires out-of-order execution model. The following polices are evaluated by our machine model.

- Fixed-size partition: the instruction stream is broken into traces at a fixed 16-instruction interval.
- Basic block partition: trace is partitioned at basic block boundary. We allow up to 3 conditional branches that can exist in a single trace. The maximum size is 16 instructions. That is, the 4th branch instruction, or the 16th instruction, whichever takes place first, will finish the current trace. In addition, we also use indirect jumps as the trace boundary since indirections cause some difficulties to maintain the thread table by the unknown target address.
- Mem-slice partition: after a minimal skip distance of 16 instructions, a memory operation will start a new trace.
- Loop partition: innermost loop iterations are parallelized. Rather than that, the rest program execution is kept sequential.
- Loop partition + basic block: in addition to loop partition, the non-loop execution and non-innermost loop execution is partitioned by basic block schemes.
During the above policies, only loop partition policy distinguishes sequential threads and parallel threads. All other policies treat all threads as parallel threads. That is, the perfect control flow knowledge, or in another words, the perfect control prediction, is implied in our execution model. Although this assumption is far from reality, it treats all policies quite fairly, except for loop partition policy, where not all threads can be executed in parallel. This unfairness is compensated by loop + basic block partition scheme. We hope all the policies can be evaluated fairly with such assumption.

We also compare the performance of the multi-threading processors utilizing different partition policies with a base machine model that simply executes instructions by an in-order and single-issue manner.

The simulation results are illustrated in two parts. Fig. a. gives the performance comparison in terms of IPC. Fig. b. shows the cycle time broken down by busy time, overhead + startup time, synchronization time, waiting for commitment time and idle time. These cycle times are normalized to CPI. The simulation results of 6 integer benchmarks and 5 floating point benchmarks are given.
compress95 (a)

- Base
- single-issue, in-order
- single-issue, out-order
- dual-issue, in-order
- dual-issue, out-order

compress95 (b)

- Idle
- Wait for commit
- Synchronization
- Overhead + startup
- Busy

si: single-issue, inorder
so: single-issue, outorder
di: dual-issue, inorder
do: dual-issue, outorder
**Image Description**

The image consists of two graphs labeled (a) and (b), which compare the performance metrics of different instruction types across various types of processor designs.

### Graph (a)

**Graph Title**: jpeg (a)

- **Y-axis**: IPC (Instructions Per Cycle)
- **X-axis**: Instruction Type
- **Legend**:
  - Base
  - Single-issue, in-order
  - Single-issue, out-order
  - Dual-issue, in-order
  - Dual-issue, out-order

**Data Points**:
- **Fixed Size**: IPC values ranging from 0 to 2.6
- **Basic Block**: IPC values ranging from 0 to 2.6
- **Mem Slice**: IPC values ranging from 0 to 2.6
- **Loop**: IPC values ranging from 0 to 2.6
- **Loop + Basic Block**: IPC values ranging from 0 to 2.6

### Graph (b)

**Graph Title**: jpeg (b)

- **Y-axis**: CPI (Cycles Per Instruction)
- **X-axis**: Instruction Type
- **Legend**:
  - Idle
  - Wait for commit
  - Synchronization
  - Overhead + startup
  - Busy

**Data Points**:
- **Fixed Size**: CPI values ranging from 0 to 1.3
- **Basic Block**: CPI values ranging from 0 to 1.3
- **Mem Slice**: CPI values ranging from 0 to 1.3
- **Loop**: CPI values ranging from 0 to 1.3
- **Loop + Basic Block**: CPI values ranging from 0 to 1.3

**Comparison**

- For both graphs, the performance metrics are compared across different instruction types and processor designs, showing variations in IPC and CPI values.
li (a)

li (b)
hyd2d (a)

- Base
- Single-issue, in-order
- Single-issue, out-order
- Dual-issue, in-order
- Dual-issue, out-order

hyd2d (b)

- Idle
- Wait for commit
- Synchronization
- Overhead + startup
- Busy

- Single-issue, in-order
- Single-issue, out-order
- Dual-issue, in-order
- Dual-issue, out-order
Based on the simulation results, we can make following observations.

1. The speedup to from multi-threading processor is limited. We compare the 8-thread unit, single-issue, in-order multi-threading processors with the base line model, the speedup is not as much as 50% in most instances. Sometimes the multi-threading model even performs worse. The single-issue, in-order multi-threading processors have 8 X issue rate, while the speedup is very limited.

2. Out-of-order execution can bring 20% - 50% speedup compared to in-order model. This can be seen from most of the simulation results in comparison with in-order and out-of-order model. In in-order execution model, if one instruction is not ready, the whole thread unit is stalled although there could be ready instructions behind the current one. The out-of-order execution can significantly reduce the synchronization time by looking for instructions out of program order.

3. Doubling issue rate can yield 80% speedup to single-issue model.

4. Fixed size, basic block and mem slice policies have better performance than rest two policies. In most cases, basic block policy performs best but the difference is rather insignificant. The loop partition policy is always the worst. Sometimes it is even worse than the base line model. There are two reasons that accounts for the poor performance of loop partition. First, loop partition only works on innermost loop iterations. The non-innermost loop iterations and sequential portion of the program will be executed in serial. Therefore, benchmarks that are not innermost loop intensive tend to exhibit bad performance on loop partition, e.g., li, perl, fpppp. Second, the parallel loop threads are likely to be serialized due to inter-thread data dependencies. Without compiler support, we can hardly identify loop index variables and loop induction variables that cause loop-carried dependencies on each iteration. Therefore, the innermost loop intensive benchmarks also perform poorly on loop partition, e.g., compress, ijpeg, swim.

5. The primary reason that accounts for limited improvement is due to the large amount of slots spent on synchronization. Because of the tight dependencies chains across thread boundaries, parallel threads are likely to be serialized. It can be observed that most of the non-busy cycles are spent on synchronization in multi-threading processors.

6. All partition policies have very low idle time except loop partition. This is because only in loop partition we specify sequential threads and parallel threads, while in all other policies we assume all threads can be executed in parallel. Therefore, loop partitioning makes high percentage of unutilized slots on waiting where innermost loops are not intensive in the program, for example, li, perl, and fpppp. In these benchmarks, it is notable that a larger portion of idle slots represent for the part where sequential execution is performed on multi-threading processor. On the other hand, in program where innermost loops are dominant, for example, compress, jpeg, swim, the idle portion is reduced since there are enough parallel loop threads that can keep thread units busy. However, the synchronization cost grows because of the tight inter-thread dependency in loop threads. This effect offsets the benefits from loop level parallelism in these benchmarks.

7. Thread overhead can cost take as much as useful computation does in total cycle time. This costly overhead implies that even if we have perfect knowledge of control data prediction thus no synchronization or misprediction can exist, we still
need to pay 50% of CPU time on creating and completing threads. Such overhead is related to thread size. In current simulation, we assume a 15-cycle overhead per thread: 5-cycle creation, 4-cycle startup, 3-cycle retirement and 2-cycle commitment. Since the thread size is around 16 instructions in most cases, such 15-cycle overhead is quite expensive to small-scale threads. It is notable that thread overhead represents a very small portion in some floating point benchmarks while using loop partition policy, e.g., swim, and tomcatv. This is because floating-point benchmarks have much larger loop body (around 1000 instructions). Therefore, a 15-instruction overhead is negligible. This observation suggest that we should either minimize the thread overhead or appropriately adjust the thread size of keep the thread overhead in a minor factor to affect the performance.

8. Imbalanced load results in more waiting cycles between retirement and commitment. In our simulation, basic block and loop + basic block policies tend to yield most imbalance thread size. As a result, the portion of waiting for commitment cycles is larger than that of rest polices. For loop partition that brings most severe problems on load imbalance, the cost is transferred to idle cycles when not all thread units are busy.

**Conclusion and Future Work**

The current study gives some useful hints to further exploit more effective partition schemes. First, due to the sequential nature of the program, the parallelization of a sequential program is likely to confront the tight dependency chain that is most likely to require serialized execution on parallel threads. Therefore, breaking the dependency bottleneck becomes a very important issue in multi-threading processors. Current study suggests out-of-order execution, multiple function units, some compiler techniques, and value prediction can be helpful to break the dependency barriers between parallel threads. We will need to employ these techniques into our machine model. Second, imbalanced thread partitions are not likely to yield optimal performance. Our results indicate some overhead of imbalanced partitions are likely to bring more waiting slots on thread commitment. Also, the loop partition is another form of imbalanced load at the sequential part.

Our results indicate that the current multi-threading machine model is not quite effective to measure the performance of different thread partitioning policies. The primary reason is due to the frequent dependencies across threads that will stall the thread execution in synchronization stage. It is very common that an early instruction in a thread must wait until the value to be passed from instructions that are located in very late portions of other threads. This will almost serialize all thread units in in-order model. This situation can be partially resolved by multiple-issue and out-of-order execution. However, the data synchronization on true dependencies still remains the major bottleneck on parallel execution. Further simulation with fully-fledged architecture is needed to evaluate the existing partitioning schemes, as well as to devise how the adaptive schemes can take effect.