計算機アーキテクチャ 139－7
（2000．8．3）

A Thread Partitioning Algorithm using Structural Analysis

NIKO DEMUS BARLI,† HIROSHI MINE,† SHUICHI SAKAI† and HIDEHIKO TANAKA†

Speculative Multithreading has been proposed as a method to increase performance of a single thread program on a Chip Multiprocessor. To achieve optimal performance, however, an effective thread partitioning algorithm is required. This paper proposes a thread partitioning algorithm using Structural Analysis, and describes design rationale behind the algorithm. Preliminary evaluation shows that a moderate performance can be achieved with a simple heuristic. However, further refinement is needed in order to achieve more flexibility and better performance.

1. Introduction

Chip Multiprocessor (CMP) approach has emerged as a promising approach for future processor architecture. CMP is constructed from several relatively simple processing units, each is capable to maintain single thread control. Compared to centralized approach such as found in superscalar, CMP has several advantages. It offers an effective way to utilize available silicon space, and at the same time, simplicity in design eases the requirement for validation phase, which has been becoming the most time consuming phase in processor design. CMP processor can naturally exploit parallelism from multithread programs or in a multiprogram workload, and well fitted to the requirements for server processors.

Despite the advantages described above, however, for CMP approach to be fully accepted, it must also be able to give high performance when executing single thread (sequential) programs. Sequential applications constitute a major portion of software today, so that it is essential for CMP to be able to execute these programs at competitive speed.

To accelerate performance of single thread applications, Speculative Multithreading has been proposed in some CMP architectures[2)3)4)5)6)]. These architectures partition a program into threads and speculatively execute them in parallel. These threads are not necessarily independent, but may include some sort of dependences (control, register or memory). The processors aggressively execute these speculative threads, and take necessary action when the speculative execution failed.

To effectively utilize CMP resources and achieve high performance with speculative multithreading, an effective thread partitioning mechanism is required. In this paper, we propose an algorithm to partition a program into a speculative threads. The algorithm statically analyze control flowgraph of a program and partition the flowgraph into subgraphs, which we will call threads. Observing that it is difficult to find an algorithm which is optimal to all classes of applications, we design the algorithm to be flexible. It uses structural analysis[7)8)] to find thread candidates, and uses heuristics to pick an optimal combination of threads. Structural analysis provides a common framework, so that different type of heuristics and thread selection methods can be applied selectively.

The rest of this paper is organized as follows. Section 2 describes related work on thread partitioning algorithm. Section 3 describes structural analysis and how we use the result of structural analysis as a common framework for thread selection mechanism. Section 4 describes current implementation of the algorithm and shows preliminary evaluation results. Finally, section 5 concludes this paper and describes future works of our research.

2. Related Works

Many CMP architectures which support some

† Graduate School of Engineering, University of Tokyo

sorts of speculative thread executions have been proposed. Each architecture has its own way to partition a single thread program into speculative threads. Multiscalar architecture use heuristics based on task size, control dependence and data dependence to statically traverse program's CFG and form speculative threads[9]. SKY architecture employs thread model that guarantees no control dependence between threads. A combination of threads which has maximal estimated gain is selected. The gain estimation uses fork distance and data dependence characteristics as parameters[10]. Hydra and DMT (Dynamic Multithreading) architecture spawn speculative threads on function invocations or loop iterations. Programmer or compiler helps the processor by marking functions or loops on which the processor will spawn speculative threads[2)12]. MUSCAT architecture does register dependence analysis, and puts fork instruction where the register dependences between current and next thread are resolved[11].

Our proposal for thread partitioning algorithm differs from previously proposed methods in that we use structural analysis to find thread candidates. It also differs in that we use a common framework provided by structural analysis to apply different kind of heuristics and thread selection algorithms, which will be selected to suit the target application the best.

3. Thread Partitioning using Structural Analysis

To run an application on a speculative multithreading architecture, we first need to identify threads. This may be achieved in software by performing a compilation step, or in hardware that support dynamic thread identification. To keep the CMP hardware simple, we choose the first approach and propose a thread partitioning algorithm described below.

3.1 Design Considerations

There are some important points to be considered when designing the algorithm. First, the algorithm should not take unreasonable long time. A brute effort to find all possibilites of thread partitioning, for example, will result in an algorithm that requires a huge amount of time to do the analysis.

Second, we need some flexibility to choose which thread selection method to use for a particular class of application. Performance gained by speculative thread execution is largely influenced by characteristics of application. Multimedia application, for example tends to have large parallelism and can be exploit with large threads. Language processing programs, on the other hand, tends to have little parallelism and may suffer from misspeculation if partitioned into large threads. While we can optimize the algorithm, to meet requirements of one class of application, the result will likely not always be suitable for all other types of applications.

Another consideration is harmony with currently available analysis algorithms. Data dependence analysis such as live register analysis, will be used extensively inside the algorithm. These types of analysis have already reached a matured point, so that it is preferable if the partitioning algorithm can use them directly.

Considering the above points, we propose a partitioning algorithm using structural analysis. The algorithm mainly consists of 2 phases. First, the algorithm analyze program's control flowgraph using structural analysis and find thread candidates. This phase reduce the number of candidates significantly, compare to a brute approach. Thread candidates are in the form of extended basic block with single entry node. This will make data dependence analysis easier since many analysis algorithms are directly applicable.

The next phase of algorithm is applying heuristics to select optimal thread combination. We will show that structural analysis provide a common framework, on which we can apply different type of heuristics in a flexible way. This will allow us to select optimal approach for different class of applications.

3.2 Thread Model

We define thread as a connected subgraph of a static control flowgraph with single entry node. A thread can be described as $G <$*Nodes, Edges, entry*$>$ where *Nodes*, *Edges* and *entry* denote set of nodes included in the thread, set of edges included in the thread, and entry node of the thread, respectively.

According to this definition, a thread may comprise a basic block, multiple basic blocks,

loop body, entire loop, or entire function. Also, logically, a thread may also comprise one or more threads. This hierarchical property of our thread model will be exploited when we use structural analysis described in the next subsection.

For our thread model, we allow control and data dependence (both register and memory) between threads. We assume a CMP hardware similar to Multiscalar[3] to execute speculative threads generated by our algorithm.

3.3 Structural Analysis

Structural analysis is originally proposed by Sharir[7] to make syntax directed method of data-flow analysis applicable to lower level intermediate code. It can discover control flow structures from arbitrary flow graph. Thus, for example, it can take up a loop and discover that it has a form of a *while* or *repeat/self loop*, even though there is no syntactical information available.

One important concept in structural analysis is that every region it identifies has exactly one entry point. The definition of region used in structural analysis is identical to the definition we used to define our thread model, so that we are able to directly treat regions found by structural analysis as thread candidates.

Figure 1 and 2 shows typical acyclic control structures (regions which do not contain any back edges) and cyclic control structures (regions which contain back edges), respectively, that can be recognized by structural analysis. The regions shown here represent the most common control structure found in program written in C language.

Proper region shown in Figure 1 is an arbitrary acyclic structure that failed to be identified by other simple acyclic cases. Since a control flow graph of program can take any possible form, it is difficult to set the algorithm to recognize all the possibilities of control structures. For this reason, the algorithm will recognize an acyclic region as a proper region if the region cannot be further reduced by any of the simple acyclic cases. Similarly, *natural loop* shown in figure 2 is a special case of cyclic region which includes loop structure that is natural but cannot be further reduced. Loop that is not natural, such as the one in figure 2 will be identified as *improper* region.

Structural analysis proceeds by examining control flow graph's nodes in postorder and try to find instances of the various region types described above. Postorder traversal is a traversal in which each node is processed after its successors. For example, if node A have two successors, B and C, in postorder traversal node A will not be processed until node B and C are processed.

After identifying a region of nodes, structural analysis will collapse and replace the region with an abstract node, reconstructing flowgraph for further analysis. This steps are repeated until the entire flowgraph collapsed into a single node.

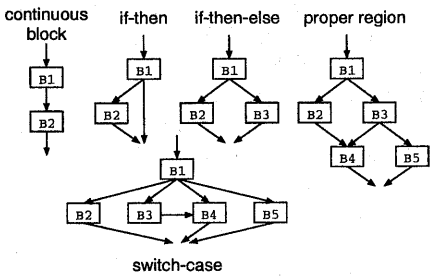

Fig. 1 Acyclic regions

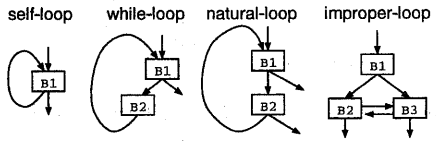

Fig. 2 Cyclic regions

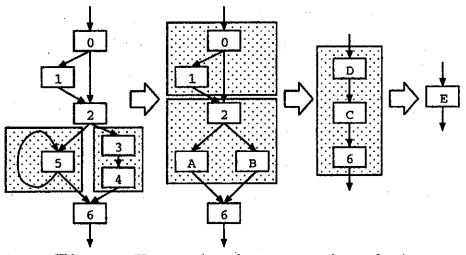

Fig. 3 Example of structural analysis

Figure 3 illustrates an example of structural analysis. Following the postorder traversal, first the analysis algorithm examines node *6* but fails to find any identifiable region. Next, it examines node *5* and finds a *self-loop* region. It

then collapses the region into node *A* and reconstructs the flow graph. Successively, it finds *continuous-block* region formed by node *3* and *4*, and collapses it into node *B*. In the same manner, *if-then-else* region formed by node *2*, *A*, and *B*, is identified and collapsed into node *C*. *if-then* region formed by node *0* and *1* is collapsed into node *D*. And lastly *continuous-block* region formed by *D*, *C*, and *6*, is collapsed into node *E*.

As shown in the example above, structural analysis gradually collapse a control flow graph of a program into a single node. During the process, the algorithm constructs corresponding control tree, which defines inclusion relation between identified regions. Since we treat the regions identified by structural analysis as thread candidates, the control tree also defines inclusion relation between the thread candidates.

Figure 4 shows the control tree of our previous example. From the figure we are able to identify that, for example, thread candidate *E*, includes thread candidates *D*, *C*, and *6* in it.

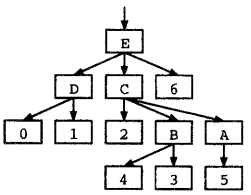

Fig. 4 Control Tree of previous example

3.4 Heuristics and Tree Coloring Rules

After identifying thread candidates using structural analysis, we can then select a combination of threads that is the most likely to give best performance. In general, since it is hard for the compiler to know precisely what will happen during execution (e.g. cache stall, branch misprediction, TLB miss etc), a compiler can at best predict and use some heuristic approaches to find the most promising threads combination.

There are some possible ways to apply heuristics to our control tree.

- Find all combination possibilities, define a heuristic function and then apply the function to find the "best" combination.
- Apply some types of heuristic in sequence, and identify one or more threads in each step.
- Combination of the above two approaches.

Regardless of what type of approach used in applying heuristic, the process of finding the right combination should guarantee that the combination found covers the original control flow graph entirely. It must also guarantee that there is no overlaps between threads. To fullfil this requirement, we defined some rules that must be obeyed when identifying thread from a control tree. We call this rules *Tree Coloring Rules*.

(1) If node X is identified as a thread, then all of X's predecessors and all of X's successors cannot exist as threads.

(2) If node X is identified as not-a-thread (i.e. there is no possibility that the node will be identified as thread), then all of X's *direct* successors that have no successor must be set as a thread.

(3) Tree coloring is finished when all nodes are checked (i.e. identified as thread or as not-a-thread).

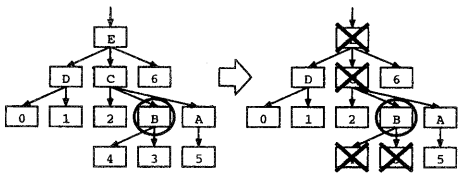

Fig. 5 Tree Coloring Rule 1

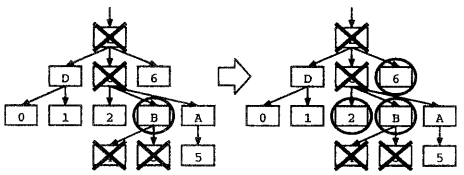

Fig. 6 Tree Coloring Rule 2

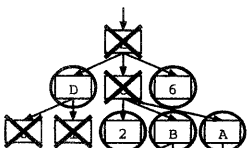

Fig. 7 Tree Coloring Rule 3

Figure 5, 6, and 7 illustrates an example of Tree Coloring Rules when applied to our control tree in figure 4. Suppose that node B is identified as thread. Then following rule no. 1,

all of B's successors (*3* and *4*) and predecessors (*C* and *E*) cannot possibly exists as threads, so that we put cross marks on them (figure 5). Next, we discover that node *6* whose direct predecessor already marked as not-a-thread, has no successor. Applying rule no. 2, we mark node *6* as a thread. In similar fashion, we also mark node *2* as a thread (figure 6). Suppose successively we identify node *D* and *A* as threads and apply rule no.1, the resulted control tree is shown in figure 7. Following rule no. 3 then we know that the process of finding threads combination is finished.

4. Implementation and Preliminary Evaluation

We implemented the partitioning algorithm as a binary annotation tool for Alpha-AXP binary. First, we build control-flow graph from the binary, then we perform structural analysis on function per function basis. For preliminary evaluation, a simple heuristic is incorporated to the annotation tool. The heuristic first find innermost loops in control tree and mark the iterations as threads. Then, for the rest parts of control tree, it finds and marks the biggest possible regions as threads.

Annotation informations generated by the analysis are used to generate execution traces, to be then analyzed by a CMP trace simulation tool. Table 1 shows the simulation parameters. Currently, we do not simulate pipeline in detail. Fetched instruction is executed as soon as its dependences are resolved and there is execution unit available.

Table 1 Trace simulation tool parameter

No of. PUs	4 Processing Units
PU parameters	Out-of-Order Superscalar 4 functional units 2 load/store units 4 instruction fetch width 64 entry instruction window 1 cycle execution for all insts.
Delay	1 cycle thread spawn delay 1 cycle comm. latency 1 cycle restart delay
Ideliazed conditions	Perfect cache Perfect branch prediction Perfect speculative store buffer

We simulated 8 programs from SPECint95 benchmark suites. The result is shown in figure 8. The evaluation shows a moderate performance, ranging from 3 to 5 IPC, can be achieved for each application. This is about 1.5 - 2 times of the IPC achieved by an out-of-order superscalar processor which has same capability as one PU (Processing Unit) of CMP.

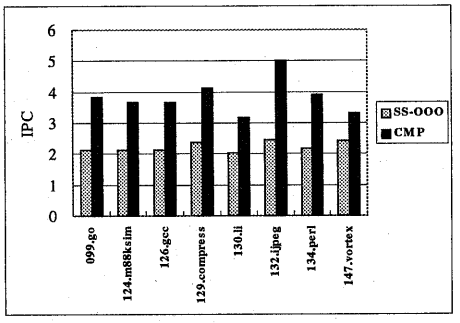

Fig. 8 Performance estimation

Figure 9 shows size distribution of threads created with our algorithm. Except for ijpeg, most of the threads created in each apllication have size of less than 20 instructions. We found that the overall thread size is small then we expected. Small threads are not only sensitive to increasing delay, but they also limit the potential of parallelism that may be exploited. For example, a 4 PUs CMP running 4 threads of 10 instructions each, will have an effective instruction window stretched for only 40 instructions, which is relatively small even when compared to window size of currently available superscalar processors.

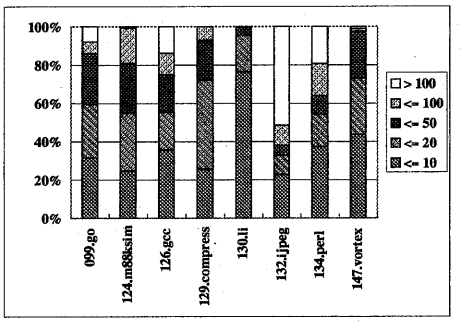

Fig. 9 Thread size distribution

Investigating problem of small threads in our algorithm, we found that it is mainly caused by restrictions imposed by function calls. Since

our algorithm and thread model assume that new thread will always be created at function boundaries, we must put thread partition whenever a function call encountered. This strictly limits possible combination of threads, resulted in many small threads created. There are however some possible solutions to the problem.

- Selectively inline functions so that bigger threads, spanning accross function calls, can be created.
- Incorporate function calls recognition into structural analysis, so that even when we must put partition at function calls, there is still enough freedom to create bigger threads.
- Add some architectural features to prevent performance degradation caused by small threads. For example, the processor can be made to switch from speculative multithreading mode to normal mode when it executes parts of programs with small threads.

5. Conclusion

We proposed an algorithm to partition a program into threads using structural analysis. Observing that it is difficult to find an algorithm which is optimal to all classes of applications, we design the algorithm to be flexible to let different types of thread selection heuristics applicable. Structural analysis used in the algorithm, combined with Tree Coloring Rules, provide a common framework for identifying threads using different types of heuristics.

Currently we develop the algorithm as a binary annotation tool targeted for Alpha-AXP binaries. Simple heuristic to identify innermost loop iterations is incorporated. Preliminary evaluation shows that, for a given parameter, moderate performance can be achieved. However, we also found that the restriction imposed by function calls, reduced freedom to create different combination of threads significantly and resulted in many small threads created.

To cope with the problem, we plan to investigate solutions as described in previous section. We also plan to develop and incorporate some more sophisticated heuristics, involving control-flow and data-flow analysis into our algorithm.

References

[1] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson and Kunyung Chang, The Case for a Single Chip Multiprocessor, Proceedings of the 7th International Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Cambridge MA, October 1996

[2] Lance Hammond, Mark Willey, and Kunle Olukotun, *Data Speculation Support for a Chip Multiprocessor*, Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 58-69, San Jose CA, 1998

[3] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar, *Multiscalar Processors*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 414-425, 1995

[4] 小林 良太郎, 岩田 充晃, 安藤 秀樹, 島田 俊夫, 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ*SKY*, 並列処理シンポジウム JSPP'98, pp.87-94, Jun 1998

[5] 鳥居 淳, 近藤 真己, 本村 真人, 西 直樹, 小長谷 明彦, *On Chip Multiprocessor* 指向 制御並列アーキテクチャ*MUSCAT*の提案, 並列処理シンポジウム JSPP'97, pp.229-236, 1997

[6] Venkata Krishnan, Josep Torellas, *A Chip-Multiprocessor Architecture with Speculative Multithreading*, IEEE Transactions on Computers, Vol. 48, No. 9, September 1999

[7] Micha Sharir, *Structural Analysis : A New Approach to Flow Analysis in Optimizing Compilers*, Computer Languages, Vol. 5, Nos. 3/4, 1980, pp. 141-153

[8] Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kauffmann, 1997

[9] T.N. Vijaykumar and Gurindar S. Sohi, *Task Selection for a Multiscalar Processor*, Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31), Nov-Dec 1998

[10] 岩田 充晃, 小林 良太郎, 安藤 秀樹, 島田 俊夫, 制御等価を利用したスレッド分割技法, 情報処理学会研究報告 97-ARC-128 pp.127-132, Mar 1998

[11] 堺 淳嗣, 鳥居 淳, 近藤 真己, 市川 成浩, 大俣 仁美, 西 直樹, 枝広 正人, 制御並列アーキテクチャ向け自動並列化コンパイラ手法, 並列処理シンポジウム JSPP'98, pp.383-390, 1998

[12] Haitham Akkary, Michael A. Driscoll, *A Dynamic Multithreading Architecture*, Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31), Nov-Dec 1998