PERFORMANCE ESTIMATION OF HOT-PATH LIMITED SPECULATIVE MULTITHREADING ON ENTIRE PROGRAM

Kanemitsu Ootsu Takeshi Ohkawa Takashi Yokota
Graduate School of Engineering, Utsunomiya University
7-1-2 Yoto, Utsunomiya, Tochigi 321-8585, Japan
email: kim@is.utsunomiya-u.ac.jp

ABSTRACT
At the present, multicore processors are ubiquitous but parallelization is often difficult in conventional parallel processing, due to the need to conservatively keep the control and data dependencies inherent in program code. In this paper, we propose a method of Speculative MultiThreading (SpMT) along hot program paths, dividing the target program code into thread codes along the hot path for each program segment and speculatively executes on a multicore processor. This method utilizes the fact that in most cases a small set of program execution paths (dynamic control flow) among a large number of possible paths within non-numerical programs is executed and raises the possibility of successful speculative thread execution. We estimate the performance achieved by the proposed method by trace-based simulation using SPEC CINT2000 benchmark programs. These results show that the method can achieve 90% of the performance of perfectly speculated execution using all program paths. The results show that highly efficient SpMT is achieved, even if we limit the target of speculative parallel execution to only the hot path.

KEY WORDS
Multicore processor, speculative multithreading program, hot path, path-based execution, trace-based simulation

1 Introduction
At the present, multicore processors are ubiquitous and greatly contribute to the performance improvement of modern computer systems. However, it is no longer easy to improve system performance by only instruction level parallelism. In server fields, processing large numbers of tasks or processes independently at the same time significantly improves performance. Especially, performance is improved by throughput-oriented computing that focuses on increasing the number of tasks or processes to be processed per time through the parallel processing of independent tasks on multiple processor cores. However, in desktop computing or mobile computing fields, because a small number of tasks may be ready to run at the same time, we cannot fully utilize the performance of multicore processors through throughput-oriented computing.

Given this situation, we face the need of latency-oriented computing that focuses on the reduction of execution time of single programs by dividing a program into multiple threads and processing them in parallel on a multicore processor. The most basic approach in latency-oriented computing is to utilize loop level parallelism. Indeed, the effect of loop parallelization is very large, especially in numerical programs. However, in non-numerical programs, the execution ratio of loop to an entire program is not high as compared to numerical programs. Therefore, we cannot speedup an entire program by only loop parallelization. To speedup an entire program, we need a parallelization technique applicable to the entire program. However, in conventional parallel processing, due to the need to conservatively keep the control and data dependencies inherent in program code, parallelization is difficult in many cases. For example, even if a dependency rarely occurs in parallel threads, it is necessary to conservatively synchronize the dependent threads in preparation for that rarely occurring dependency. Thus, the opportunities for parallel execution are lost. In particular, for non-numerical programs, many possible events can lead to the loss of parallel execution, due to their inherent complex dependencies, and thus the speedup of a program is likely to be low.

On the other hand, the behavior of program execution is biased. By taking advantage of this bias, it is possible to break partial program dependencies. For example, if dependency rarely occurs between parallel threads, speedup can be achieved by parallel execution on the assumption that there is no dependency. Of course, in the case that the dependency actually occurs between parallel threads, the performance might be degraded, since re-execution after rollback of the program state is required. However, if the frequency of re-execution is small, the opportunities for parallel execution increase and the performance is improved on the whole. This method, called Speculative MultiThreading (SpMT), is required to run a program in less time than the critical path within the program. In the present situation, where further speedup by utilizing instruction level parallelism within a single program is difficult, it is considered that SpMT on multicore processors becomes inevitable for performance improvement of a single program. Various researches on SpMT have been reported [1, 2, 3, 4] to boost the performance of latency-oriented computing on a multicore processor.

The purpose of this research is to speedup non-numeric programs by SpMT on a multicore processor. We
take into account the fact that in most cases a small set of program execution paths (dynamic control flow) among a large number of possible paths within non-numerical programs are executed. That is, a small number of paths are executed frequently in each part of the program. We refer to the most frequent execution path as the hot path. In this paper, we propose a method of SpMT along a program hot path. This method divides the target program code into thread codes along the hot path for each program segment and speculatively executes on a multicore processor. Then, we estimate the performance obtained by the proposed method by trace-based simulations using SPEC CINT2000 benchmark programs.

The remainder of this paper is organized as follows. Section 2 presents the importance of the hot path and describes the method of hot-path limited speculative multithreading. Section 3 describes how our method is applied to an entire program. Section 4 shows the performance estimation results. Section 5 describes related research, and Section 6 concludes this paper.

2 SpMT along hot program path

Various methods and architectures on SpMT, such as Multiscalar[1] and superthreaded architectures[2], have been studied. They have the similarity that a program is divided into code fragments along the program’s control flow and each code fragment is speculatively executed in parallel. Compatibility with existing programs is an advantage. Namely, thread-level parallelization of the existing program is easily performed. Therefore, in this paper, we discuss the method of SpMT, which divides a program along its control flow into threads to execute in parallel.

Bias, as the most important behavior of a program’s control flow, exists in the dynamic control flow (that is, path). In our previous research, we confirmed that, although a single program can contain a large number of program paths, only a few paths are actually executed [5, 6]. This behavior allows us to appropriately select the speculative threads that likely succeed in execution speedup. In addition, by limiting the control flow to the hot path (the frequently executed flow), it is possible to simplify the handling of dependencies between threads to reduce the cost of generating parallel thread code. Furthermore, each thread executes a straight line code without a conditional branch from the start to the end of the path (as long as the path is the actually executed path). Therefore, each thread code can be handled as a large basic block, and optimization techniques such as instruction code scheduling and elimination of useless code[7] can be easily applied. Here, we discuss the hot path and SpMT. Prior to our discussion, we explain the Ball-Larus (BL) path and introduce the concept of a path group.

The BL path is defined such that a path is terminated by a backward branch[8]. The BL path is advantageous in that each loop iteration within a program can be naturally extracted based on a simple and clear strategy. Needless to say, the program loop is the most important target of thread-level parallelization. Fig.1 shows an example of the BL path. In this figure, rounded rectangles indicate basic blocks and arrows indicate transitions between basic blocks. The control flow graph (CFG) in the figure includes one loop, containing one conditional branch. In this CFG, eight possible paths exist. For each path, the sequence of basic blocks is shown on the right. For example, paths p1 and p2 start from the basic block at the top of the CFG and reach the endpoint of the graph through a backward branch at the end of block E. In the loop, path p1 passes through block C and path p2 passes through block D. On the other hand, paths p3 and p4 also start from the top of the CFG, but these paths are terminated by the backward branch at block E. Based on the definition of the BL path, a path is terminated by the backward branch when the branch is taken, whereas a path continues without being terminated by the backward branch when the branch is not taken.

Next, we introduce the concept of a path group. A path group is a set of distinct paths that share the same entry point. In the example of Fig.1, paths p1, p2, p3, p4 and paths p5, p6, p7, p8 are in the same path group, respectively, since they share the same entry point of the basic block. Two path groups exist, and each path group includes four paths. The path group is a concept for collectively handling the paths sharing the same starting point. When program execution reaches a path group, one of the paths within the path group is executed. It is possible to increase the success ratio of speculative execution by selecting the most likely path within the path group. Of course, in some cases, only one path belongs to a path group. In this case, that path is certainly executed.

On the left side of Fig.2, the frequency distribution of path groups in the SPEC CINT2000 programs is shown. The horizontal axis lists the number of paths in the path group, and the vertical axis shows the frequency of the path groups for each number of paths. For ease of comparison between programs, the frequency of the path group is normalized to the total number of paths in each program 1.

1 Value 1 means the total number of paths.
In addition, this figure only shows the frequency distribution with respect to the number of paths up to 10, since the frequency of path groups containing more than 10 paths is quite small. This result shows that a large portion of path groups in most programs have only one path, except for 197.parser. This means that the path to be executed can be mostly determined by each path group in many cases. In other words, we can expect a high possibility of successful execution of speculative thread codes by selecting a single path for each path group. Meanwhile, in some cases, such as 197.parser, the path group does not determine the path to be executed. In this case, the number of possibly executed paths is limited to two or three at most. Thus, it is expected that we can select the most likely executed path with high confidence, since it is sufficient to select one from the small number of paths.

To clarify the possibility of the path selection with high confidence, we show the execution ratios of hot paths (the most frequently executed paths) for each entire program. We refer to this execution ratio as path coverage. On the right side of Fig.2, the path coverage for each SPEC CINT2000 program is shown. This result shows that hot paths can cover more than 80% for most programs (except 175.vpr and 300.twolf), and can cover 84% of an entire program on average. This means that the SpMT and selecting the most frequent path can cover the execution of more than 80% of an entire program. Since selecting the most frequent path is easy by path profiling or a similar method[8], this SpMT method is highly feasible and allows us to achieve relatively high performance.

Fig.2 shows that most path groups contain single path. This means that only one path is actually executed and does not mean that the path group really contains only one path. More than one paths might potentially exist within the path group and all other paths (except one) might be not executed actually. That is, a path group might contain multiple possible paths even if only one path is actually executed.

In the SPEC CINT2000 programs, the ratio of paths that really contain only one path is not high, the percentage is 27% on average. Thus, typical path groups possibly contain multiple paths in non-numerical programs.

Based on the above discussion, we propose a method of SpMT, hot-path limited speculative multithreading, that executes speculative thread codes along hot paths in parallel to speedup the sequential program. We explain this method by using Fig.3. The CFG on the left side of this figure shows a loop with conditional branches. In this loop, three BL paths start from the basic block A, and we assume that the hot path is the path with basic blocks A, B, C, and G, since hot paths can be determined by path profiling beforehand. In advance, the basic blocks A, B, C, and G are collectively optimized to form a speculative thread code. For convenience, we refer to this code as hot-path code A, meaning that the thread code along the hot-path starts from block A. Hot-path code A is optimized by removing the redundant and useless code that is not necessary for executing basic blocks A, B, C, and G. Therefore, it can be executed in shorter steps than can the original loop iteration code.

On the right side of Fig.3, we show how the parallel execution is performed. During program execution, hot-path code A is executed whenever the execution reaches the basic block A that is the top of the loop. Each time the loop iteration is repeated, the execution of hot-path code A is started successively as a speculative thread to form the speculative parallel execution. As long as the execution of the hot path is followed, speedup is achieved by speculative parallel execution. This parallel execution is speculative and thus, when a path other than the hot path is executed, it is necessary to cancel the speculative execution and the correct execution must be performed after the recovery of the thread states. Since it affects all subsequent threads after the failed speculative one, all execution of all subsequent threads must be canceled. Therefore, the parallel execution is stopped and the performance is degraded if the
speculative execution frequently fails. The failed speculative thread restarts the correct execution by using the original sequential code. Then, parallel execution is established again after hot-path code $A$ is started one after another on each processor core.

To reduce the impact of the runtime overhead caused by the control of parallel execution, large-sized threads are desirable. However, as the thread size becomes large, the occurrence of data dependencies between threads possibly increases. Furthermore, since the continuous occurrence of a long path is rare, the threads before and after the thread executing the long path tends to become shorter. Thus, the load balance between threads is severely corrupted, and the execution performance becomes highly degraded. Therefore, we apply an upper limit to the thread size so that we do not reduce the opportunities for parallel execution. In this paper, when the number of basic blocks within a BL path exceeds 32, the path is split into small paths whose size is less than the upper limit. However, we should consider the fact that the above method is not necessarily appropriate (or upper limit value) to limit the path length. Consideration of an appropriate method of restriction is our future work.

3 Hot-path limited SpMT on entire program

In the previous section, we explained the method of SpMT along the hot path. This method can be applied to the entire program. Namely, this method decomposes the entire program into threads along the hot BL path and speculatively executes the decomposed threads in parallel.

Fig. 4 shows the thread decomposition of an entire program along the hot path. In this figure, rounded rectangles indicate basic blocks, broken-line rectangles indicate threads, and arrows indicate transitions between basic blocks. As shown in the figure, one thread is a collection of multiple basic blocks along the control flow. The right side of the figure shows the parallel execution of the decomposed threads. In this example, a program is decomposed into four thread codes along the dynamic control flow, and these thread codes are speculatively executed in parallel.

Fig. 5 shows the control of speculative thread execution. Each thread speculatively starts its execution in program order. Parallel execution is established by the subsequent initiation of succeeding threads of the restarting thread. The execution for each thread should be performed in program order to obtain the same execution results as the original sequential program code. To simplify the control of threads, the termination process of each thread is also performed in program order. In other words, the termination of a thread is suspended until all of the preceding threads are terminated. In this research, we adopt a policy of in-order thread termination, such as Multiscalar[1] and superthreaded architectures[2], since the control of the thread execution becomes simple and the quantity of hardware required for implementation becomes low.

In our hot-path limited SpMT, it is sufficient to keep only the dependencies required for execution of the hot path.

![Figure 3. Hot-path limited SpMT](image3)

![Figure 4. Thread decomposition and parallel execution of entire program](image4)

![Figure 5. Control of speculative thread execution](image5)
path; so, we do not need to keep other dependencies. However, this does not mean that all the dependencies between threads are eliminated. Data dependencies exist between threads and they must still be handled correctly. Two alternatives are available for handling the data dependencies between threads. One is to perform synchronous communication between threads, and the other is to execute threads without synchronous data communication and to confirm whether data dependencies actually exist between threads.

In addition, depending on the target multicore architecture, two alternative methods are available for data communication between threads. One method uses register access, and the other uses memory access. Multiscalar architecture allows both communication methods, whereas superthreaded architecture allows only data communication via memory access. The cost of data communication using memory access tends to be higher than that using register access because the instruction code for address calculations is always accompanied by data communication using memory access. Therefore, data communication using register access is advantageous with respect to performance. However, data communication using memory access is also necessary in the case that several data dependencies occur between threads, because, in general, few registers are available. As discussed above, we assume that both methods of data communication between threads are allowed.

The handling of data communication should differ depending on whether data communication is performed by register access or by memory access, because these data communication methods differ with respect to how data dependencies are determined.

For register access, we can statically determined whether a data dependency exists between threads by inspecting the register number. On the other hand, we cannot statically and completely determine the data dependencies between threads, because the operation for memory access includes address specification by register indirect reference. In other words, a difficulty similar to that encountered in pointer analysis occurs. Therefore, in the case of data communication by memory access, it is necessary to conservatively perform synchronous data communication on all of the memory accesses that may be dependent between threads. This causes an unnecessarily long waiting time for synchronization and may degrade performance.

Based on the above discussion, synchronous data communication between threads is considered to be the best method for data communication by register access. On the other hand, for data communication by memory access (except in the case that data dependencies are definitely determined and are not ambiguous), asynchronous data communication is considered to be advantageous with respect to performance. In asynchronous data communication, thread execution is continued despite the existence of potentially dependent data and the possible existence of data dependencies is inspected later during the run time. When a data dependency is definitely determined to be actually dependent between threads, thread execution using the old (and incorrect) data should be canceled, restored to its initial state, and re-executed from the start point. In Multiscalar architecture[1], the data dependencies are handled in the communication by memory access. On the other hand, superthreaded architecture[2] adopts synchronous communication for memory access, and the cost of instruction codes required for data communication is high. Furthermore, the execution performance may be degraded because in a number of cases, instruction codes are added for actually unnecessary data communication in preparation for data dependencies that cannot be determined statically. Therefore, we adopt the policy used in Multiscalar architecture for data dependencies.

When decomposing the entire program into threads, two alternatives are available and they depend on whether the program is decomposed at the procedure, i.e., subroutine and function, boundaries. When the program is decomposed at the procedure call, data dependencies might exist between two adjacent threads (one is the caller, and the other is the callee) in the form of arguments and return values. These dependencies might degrade the performance. Therefore, in this case, two paths, created by decomposition at the procedure boundary, should not be executed as two separate threads. On the other hand, opportunities for parallel execution decrease, because the number of threads decreases when the program is not decomposed at procedure boundaries. In other words, opportunities for performance improvement by parallel execution decrease.

Based on the above discussion, the actual behaviors of practical programs should be examined to determine whether a program should be decomposed at procedure boundaries. As demonstrated in the performance evaluation in Section 4, the performance for the case of decomposition at procedure boundaries is high. Therefore, we assume that programs are decomposed at procedure boundaries.

A problem may occur when the program is decomposed at procedure boundaries when saving and restoring register data and the updating of the stack pointer and the frame pointer upon procedure call become data-dependent between threads. (Note that we assume the multithreaded code is generated by re-using the original single-thread code.) Fig.6 shows the data dependencies and data flow when the program is decomposed at the procedure boundary. Each speculative thread is executed on a separate processor core. (This means that the instance of the register is separated.) Therefore, with respect to saving and restoring register data upon the procedure call, data dependencies may occur between threads, but the dependency can actually be removed. In the figure, for register r1, threads i, i+1, and i+2 are mutually data-dependent. Synchronous data communication between threads i+1 and i+2 should be performed to transfer the correct data. However, since data dependency on register r1 exists between the two threads, this dependency actually does not need to be preserved, because the data of register r1 from thread i to thread i+1 and the data from thread i+1 to thread i+2 have
the same value. Thus, the value of register r1 from thread i+1 does not need to be transferred to thread i+2. Instead, thread i must transfer the value of register r1 to thread i+2.

In addition, as in the above case, the values of the stack pointer and the frame pointer from thread i+1 are not transferred to thread i+2; instead thread i transfers these values to thread i+2.

As shown above, when the decomposition is performed at the procedure boundary, we can increase the opportunities for parallel execution by removing some data dependencies between threads. Indeed, by our preliminary experiments[9], we have confirmed that the performance is improved by removing some parts of the dependencies at the procedure call.

4 Evaluation

In this section, we estimate the degree of performance improvement by hot-path limited SpMT on an entire program by using practical non-numerical programs.

The performance is measured by trace-based simulation. For this evaluation, we use nine programs from the 12 programs of SPEC CINT2000[10]. Since the remaining three programs did not run on the simulator, we could not use them for this evaluation. These nine programs are executed on the processor simulator SIMCA[11] to generate instruction trace data, and it is processed by our performance estimation tool for hot-path limited SpMT.

The number of execution steps is calculated assuming that both instruction fetch and data access to the cache memory are perfectly hit. In addition, it is assumed that one instruction is executed and completed at every step in each thread, and we use the test data set as input data for each SPEC CINT2000 program. We assume that the thread initiation, termination, and rollback process on the speculation fail is completed in one step, respectively. Furthermore, data communication between adjacent threads is performed in one step.

The purpose of above assumption is to estimate performance under the ideal condition for thread control. Here we discuss its feasibility. As for thread initiation, our method predetermines a path candidate for speculative execution in each path group. That is, the hot path in each path group is selected as the target of speculative execution by using path profiling information. When a path in a path group is executed, the path executed next is determined as the hot path in the next path group, except for the paths at the first stage of program execution. Therefore, we consider that a thread can be initiated in one step by preparing the initiation of the succeeding thread during the execution of preceding threads. As for thread termination, the most largest process at thread termination is to commit the execution results by speculation. As for the contents of registers, we consider that commitment in single step can be realized by using multiple sets of registers. That is, by switching registers from one for speculative execution to another one for holding execution results until the completion of commitment process. As for memory data, we consider that commitment in single step can be realized by using cache memory as buffer for speculative data. Similar mechanism to the Speculative Versioning Cache (SVC)[12] allows us to commit speculation result in single step. As for rollback process, by using multiple sets of register and similar mechanism to the SVC, rollback process in single step can be realized. As for register contents, the content of register at thread initiation is stored in a set of register, from which the content is reloaded to the register for speculative execution when the speculation fails. As for memory data, squashe operation of the SVC allows us to rollback the speculative memory data in single step.

The performance is evaluated using two, four, six, and eight processor cores. In addition, we evaluate the performance in both cases of the program is decomposed or is not decomposed at procedure boundaries to clarify the effect of the degree of decomposition at procedure boundaries on the performance, as is discussed in Section 3.

In addition, for the purpose of comparison with the ideal case, we measure the performance in the case that all paths are prepared to execute and their execution is always successful based on the perfect path prediction[9]. In hot-path limited SpMT, the penalty cost rises when speculation fails because of the execution of a different path from the hot path. Therefore, the performance might be degraded as compared to the performance in the case that all the speculative threads are successfully executed. We clarify the degree to which the performance is degraded.

Fig.7 shows the speedup ratios for each SPEC CINT2000 program. The horizontal axis lists the program. The last data group on the right (geo.mean) shows the average speedup ratios (geometric mean). The vertical axis shows the speedup ratio relative to single-thread execution for each program. In other words, a value of 1 indicates that the performance is equivalent to the single-thread performance. In the figure, hot+dec represents the performance in the case that the program is decomposed at the procedure boundaries, and hot+nodec represents the performance in the case that the program is NOT decomposed at the procedure boundaries. In addition, perf+dec indicates the performance in the ideal case that all paths are prepared to execute and are successful based on the perfect path prediction. perf+nodec indicates the performance
in the case that the program is not decomposed at procedure boundaries and other conditions are the same as those in perf+dec. (In the perf+dec case, the program is decomposed at procedure boundaries.) In this evaluation, the instruction sequence of each thread code is the same as that of the original single-thread code, and optimizations (such as instruction scheduling) are applied to the thread code assuming single-thread execution. Therefore, we should note that the speedup ratio is considered to be lower than the actual performance, and the ratio is underestimated.

Comparison of hot+dec and hot+nodec reveals that the performance of hot+dec is higher than that of hot+nodec in the case of four or more processor cores, except for 164.gzip and 174.vpr. Thus, in these cases, performance is improved by decomposing the program at procedure boundaries. In the case of two cores, the performance of hot+nodec is higher than that of hot+dec in half of the programs. However, on average, the difference between hot+dec and hot+nodec is quite small. Therefore, program decomposition at procedure boundaries totally improves the performance.

For the average speedup ratio (geometric mean), the results indicate that a speedup of 1.25 for two cores, 1.63 for four cores, 1.83 for six cores, and 1.94 for eight cores can be attained.

Next, we discuss the performance of hot+dec and perf+dec. In Section 2, we showed that the hot path occupies 80% of the execution of the entire program. Therefore, we expect that we can achieve a performance of 80% of the performance in the ideal case by the proposed method. The ratio of the performance of hot+dec to that of perf+dec is 84% at minimum, 97% at maximum, and 90% on average, respectively. From this, we can see that the ratio of speedup overcomes the execution ratio of the hot path. That is, highly efficient SpMT is achieved, even if we limit the target of speculative parallel execution to only the hot path.

Overall, performance is improved by increasing the number of cores. However, it is not proportional to the number of cores. The reason why speedup is restricted is that parallel execution is disturbed by data dependencies between threads. Tab.1 shows average waiting time (steps) of threads for each case of the number of processor cores. Waiting time is normalized by the average steps of thread execution except for idle time. This result indicates that the waiting time becomes longer as the number of cores increases. By increasing the number of processor cores, performance is improved but the average waiting time is long. Therefore, speedup proportional to the number of cores is not achieved. The reason lies in the machine instruction codes used for the evaluation.

In this evaluation, we estimated the performance by using machine instruction codes that are optimized for sequential single-thread execution. Since those codes are optimized without taking data dependencies between threads into account, the waiting time caused by the data dependencies is not small. Therefore, although the performance is improved, it is difficult to achieve the improvement proportional to the number of processor cores. We consider that performance can be significantly improved by applying instruction code scheduling designed for parallel execution. Performance estimation by using machine codes optimized for parallel execution is our future work.

### 5 Related research

Similar research attempting to achieve speedup through thread-level parallel execution on an execution path used a trace processor[13]. The trace processor executes each trace (a physical sequence of instructions extracted from a dynamic control flow) as a thread on each independent execution engine (that is approximately equivalent to the usual processor core) while decomposing the dynamic control flow into short sequences of instructions by using the trace cache. Since each trace is extracted based on the trace cache, the trace processor has the problem of an upper limit on the number of instructions within a single trace because of the limited hardware resources. (In [13], the length of a thread code is limited to at most 16 instructions.) Our method has essentially no such limitation. A path (a logical sequence of instructions) does not need to be decomposed by the limited amount of hardware resources. Our method also has an upper limit on the thread size, but its purpose is to avoid the severe load imbalance between threads to increase the efficiency of parallel execution. The restriction does not come from a limited amount of hardware.

Another similar research study on extraction of speculative threads based on the hot path is the ASTEX system[14]. The ASTEX system is software that extracts speculative threads in C programs for embedded systems by using profiling and hot path information. The ASTEX system adopts a speculative model that does not assume shared memory. In other words, each thread is executed on a processing element with distributed (non-shared) memory. Since communication between threads is explicitly controlled with software code and data movement between threads is necessary before and after the parallel execution, the overhead of communication tends to be high. Therefore, the ASTEX system must precisely identify a minimum range of mem-

<table>
<thead>
<tr>
<th>program</th>
<th>number of processor cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.57</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.16</td>
</tr>
<tr>
<td>176.gcc</td>
<td>0.46</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0.25</td>
</tr>
<tr>
<td>197.parser</td>
<td>0.44</td>
</tr>
<tr>
<td>254.gap</td>
<td>0.29</td>
</tr>
<tr>
<td>255.vortex</td>
<td>0.45</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0.17</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.31</td>
</tr>
</tbody>
</table>
Figure 7. Speedup ratios of SPEC CINT2000 programs
ory access and requires additional profiling information on memory accesses to avoid unnecessary data movement between threads. In addition, that research focuses on exploring thread extraction and does not show actual performance evaluations.

6 Conclusion

In this paper, we estimated the performance of the hot-path limited SpMT on an entire program to clarify the degree of overall performance improvement in non-numerical programs. First, we showed the important behavior on the program control flow: the hot path (the most frequently executed program flow) covers more than 80% of the execution of an entire program. Next, based on the behavior, we showed the hot-path limited SpMT on an entire program. Here, hot-path limited SpMT decomposes the entire code of the target program into threads based on the BL path and executes these threads speculatively in parallel.

Then we estimated the execution performance by trace-based simulation on SPEC CINT2000 benchmark programs, which represent practical non-numerical programs. The performance estimation results indicate that an average speedup of 1.94 can be attained by using a multicore processor with eight cores, in comparison with sequential execution on a single-core processor. This corresponds to 90% of the performance of perfectly speculated execution using all program paths. That is, highly efficient SpMT is achieved, even if we limit the target of speculative parallel execution to only the hot path.

In the future, we intend to evaluate the performance when the speculative thread code is fully tuned by an optimization method, such as instruction code scheduling specialized for parallel execution. In this paper, since the instruction sequence of each thread code is the same as that of the original single-thread code and the optimizations applied to each code are not designed for parallel execution but rather for single-thread execution, many improvements are possible for the performance for parallel execution.

Acknowledgment This research was supported in part by Grant-in-Aid for Scientific Research ((C)24500055, (C)25330055) and Young Scientists (B)25730026) of Japan Society for the Promotion of Science (JSPS).

References


