Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms

Fredrik Warg and Per Stenström
Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{warg, pers}@ce.chalmers.se

Abstract

This paper considers program modules, e.g. procedures, functions, and methods as the basic method to exploit speculative parallelism in existing codes. We analyze how much inherent and exploitable parallelism exist in a set of C and Java programs on a set of chip-multiprocessor architecture models, and identify what inherent program features, as well as architectural deficiencies, that limit the speedup. Our data complement previous limit studies by indicating that the programming style – object-oriented versus imperative – does not seem to have any noticeable impact on the achievable speedup. Further, we show that as few as eight processors are enough to exploit all of the inherent parallelism. However, memory-level data dependence resolution and thread management mechanisms of recent CMP proposals may impose overheads that severely limit the speedup obtained.

1. Introduction

While instruction-level parallelism has been a good source to boost performance of processors over a long period of time, this source is now getting exhausted. The major reasons are the difficulties in supporting huge instruction windows as well as in speculating beyond control-flow dependences. As a remedy, several recent papers have proposed support for speculative thread-level parallelism (STLP) in the context of chip-multiprocessors (CMPs) [6, 16]. A chip-multiprocessor with support for thread-level data speculation allows programs partitioned into threads to correctly execute in parallel even if it is not ascertained that the threads are indeed data independent. If it turns out that a data dependence violation occurs, the speculation support will detect it which permits the speculation system to abort and re-execute the threads in a way that respects sequential semantics.

The most popular form of STLP to exploit has been loop-level parallelism. While impressive parallelism can be obtained in numeric applications with loops that contain few loop-carried dependences, the poor parallelism coverage or lack of do-all loops in general integer applications severely limit this approach [12]. On the other hand, module-level parallelism, i.e., parallelism across function, procedure, or method invocations, is potentially a more general and useful form of STLP. First, it is very simple to identify the thread boundaries; new threads are created at module invocations, and terminated when they reach a return. Second, we avoid the control dependence problem we encounter in, for instance, loop-level speculation. Presumably most importantly, however, is that modules are used frequently as the key abstraction mechanism in object-oriented programs in particular but also in imperative programming styles.

The first goal of this paper is to understand to what extent the programming style – imperative versus object-oriented – affects the *inherent* speculative module-level parallelism. We do this by considering a set of C and Java programs and carry out a speedup limit study assuming an idealized machine model. This model has an infinite number of processors, it supports perfect prediction on return as well as memory values, and it imposes no overhead on thread management or inter-thread communication. While Oplinger *et al.* [12] present a limit study based on a similar idealized machine model, they didn't consider Java programs.

The most important result we gained from the experiments on the idealized model is that there is a fair amount of module-level parallelism in C and Java programs. The question is how to best exploit it in terms of appropriate architectural support. We separate out a number of concerns through a series of successively refined architectural models as follows. The first issue we study is to what extent data dependences between threads (on return or memory values) limit the speedup obtained. More effective encapsulation of data in objects would speak in favor of an object-oriented style of programming. It is interesting to see whether this indeed will result in fewer data dependences and higher speedup limits when comparing C and Java programs.

Another motivation is to see whether simple value prediction schemes suffice or research into more sophisticated value prediction schemes are warranted. A third issue that we address is how much machine resources are needed to exploit the inherent parallelism. We address this issue by studying the speedup limit as a function of the number of processors and again whether the expected heavier use of modules in object-oriented programs would lead to more scalability. Finally, we also address to some extent how thread management overheads impact on the achievable speedup to see whether research into more effective support is warranted and what this support should target. One interesting aspect is how well the granularity of parallelism in terms of common module sizes matches the overheads incurred in recent CMP proposals. While [12, 11, 2] have also studied the potential of module-level parallelism in C and Java programs on CMP platforms, none of them has explicitly compared the nature of the module-level parallelism inherent in C and Java programs.

The main contribution of this paper is the insights into the inherent and architectural limits on the speedup for imperative versus object-oriented programs in a single consistent framework. Our most important findings are:

- Overall, we didn't notice any significant qualitative differences between C and Java programs suggesting that the programming style has a minor effect on the amount of parallelism to be exploited.
- The inherent module-level parallelism in applications is typically not more than four to eight suggesting that small-scale CMP or multi-threaded cores are enough to exploit all of the available parallelism.
- Most of the codes do not benefit from more advanced return value-prediction schemes than stride and last-value prediction suggesting that current predictors fare pretty well.
- The granularity of modules typically don't match the overheads in recent CMP proposals. In addition, the accuracy of memory value prediction schemes is a major inhibitor to decent speedups. This suggests that more research into better machine models and memory-value prediction schemes are warranted.

In the next section, we introduce the execution model and the series of architectural models used to identify the limits on module-level parallelism. Then in Section 3, the experimental setup along with the benchmark applications used are introduced. The experimental results are provided in Section 4. We put the work in perspective of related work in Section 5 along with an outlook before we conclude in Section 6.

2. Execution and architectural models

In this section, we first introduce the execution model as seen by the software and then introduce the machine models used to identify what the limits on achievable speedup assuming the execution model are.

2.1. Module-level execution model

The true advantage of module-level parallelism lies in its simplicity. In its simplest form, a new thread is spawned at each module (i.e., function, procedure, or method). To respect sequential semantics on a module invocation, the old thread of control executes the module, whereas a new thread is spawned that speculatively executes the code after the module call as shown in Figure 1. If another module call is encountered, a new speculative thread that executes the continuation of the module will once again be created. In order to respect sequential semantics, the new thread will be more speculative than the thread from which it was created (i.e. it would execute after the original thread in sequential execution), but retain the same relationship as its parent with respect to all other speculative threads. All threads must commit in sequential order; in other words, a thread cannot commit until all less speculative (earlier in sequential execution order) threads have already committed.

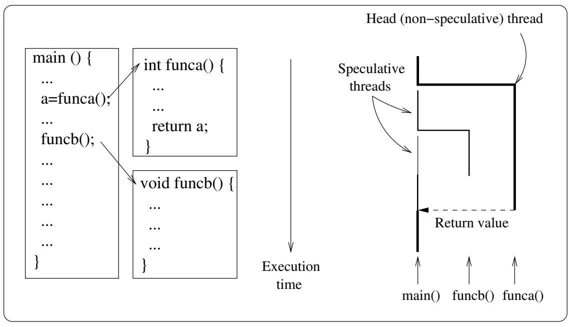

Figure 1. Execution model

2.2. Architectural models

The four models introduced gradually encounter more of the architectural limitations associated with recent CMP proposals. For each model, we first specify what limitations it encounters and then discuss what issues will be addressed with it.

Model 1: Inherent module-level parallelism

A speculative thread will successfully terminate as long as no data dependences are violated with threads that would precede it according to sequential semantics. Then the upper bound on the speedup is dictated by the control dependences between subsequent module invocations. We wish to understand the scalability of module-level parallelism in terms of how severely the control dependences set in. The first model therefore assumes a machine with an infinite number of processors and that no data dependences will be violated and cause rollbacks. In addition, thread management and inter-thread communication costs are zero.

This model provides important insights into the difference of imperative and object-oriented programming styles. One hypothesis is that an object-oriented style of programming would lead to more scalability in exploiting module-level parallelism. This is one of the hypotheses we will test using this model.

Model 2: Impact of data dependences

With this model, we are interested in how data dependences between threads limit the achievable speedup and whether proposed support in terms of forwarding and value prediction in the recent literature is enough.

There are two classes of data dependences: flow and name dependences. In this as well as in the subsequent architecture models, we assume that name (anti- and output) dependences can be resolved through renaming. In CMPs with speculation support this is done by keeping speculative state in the cache until the thread can commit, which involves flushing the speculative state back to the memory. As Steffan and Mowry have found [16], for the small-grain threads usually considered for CMPs, the available cache space seems sufficient to host the speculative state created.

On the other hand, flow dependences may have a severe impact on the achievable speedup through module-level speculation since a data dependence violation will result in a rollback. If a thread has computed a value before a more speculative thread reads it, the most recent value will be forwarded to the more speculative thread; but, if the value is computed after the more speculative thread performs the read, a flow dependence violation occurs. After a violation, the more speculative (violating) thread will rollback execution in order to maintain correct sequential execution.

The model we assume is capable of perfect rollbacks, which means that the thread causing the violation will be able to restart execution exactly at the load instruction causing the violation. However, threads started by the violating thread after the erroneous instruction are squashed.

Flow dependences take two forms: flow dependences through memory and return values. To separate out the relative frequency of each category, we experiment with six alternatives:

- Heap accesses have either (1) perfect value prediction or (2) none at all. Perfect value prediction means that the prediction is always correct, and consequently we never get any memory-bound dependence violations.
- Value prediction for return values comes in three flavors: (1) Perfect return value prediction (RVP) is once again always correct; (2) stride RVP is supported by a table storing a last value and a stride value for each procedure; the table is of unbounded size. RVP buffers are updated in execution order, which is not necessarily in the same order as in the sequential execution. Additionally, it might happen that a finished thread updates the value predictor and then gets squashed, resulting in predictor pollution; one could say that the value predictor is speculatively updated. (3) The third option is no return value prediction.

With this model, we are able to answer questions related to the relative importance of memory versus return value flow dependence violations and how they relate to the programming style. One hypothesis would be that object-oriented programs tend to better encapsulate memory-bound flow dependences whereas dependences caused by return values become more critical. In addition, it is possible to pinpoint whether it would make sense to focus future research on more sophisticated value prediction schemes for STLP.

Model 3: Impact of limited processing resources

While the scale of CMPs will increase with increased integration, it may not make sense to charge too many resources to thread-level parallelism in trading off number of processors versus issue-width for example.

In the third model, we study to what extent the number of processors limit the speedup. When the number of available threads exceed the number of processors, priority will be given to threads based on sequential execution order. New threads with higher priority will preempt more speculative threads if needed.

One problem noted in the Hydra project [6] regards the speculative state stored in the caches. To avoid having to save the cache state, it is not possible to preempt a speculative thread until all the preceding speculative threads have committed. This may severely limit the speedup obtained for module-level speculation. If preemption is impossible, a new thread cannot be created when all processors are in use, unless a more speculative thread occupying one of the processors is squashed, wasting the work it has already done. An even more serious consequence would be load-imbalance problems. If a large thread is running, completed more speculative threads can neither commit, nor yield the processor, and therefore the processor will remain idle until the large thread finishes. We do not, however, impose this limitation as our aim is to establish an upper-bound on the available parallelism. While one would have to address this issue, it does not appear as a hard problem.

Model 4: Impact of thread-management overhead

In the preceding models, we have assumed that threads can be spawned, committed, and rolled back in zero time. On recently proposed CMPs such as Hydra, the overheads imposed by these operations are not negligible. To what extent the overheads have a significant impact on the speedup obtained is strongly connected to two application parameters: the number of flow dependence violations and the module granularities. Our goal with this model is to factor in these overheads to identify what mechanisms would have to be further researched to come up with machine models better adapted to module-level parallelism.

3. Methodology and benchmarks

In this section, we first explain how the simulations were done, and then present the benchmarks used in our experiments.

3.1. Simulation tools

All results presented in the following sections are obtained from our trace-driven simulation tool. The simulation tool implements all architectural models described in the previous section. The tool runs a program much as it would be run on a real machine with speculation support, that is, threads are run in parallel with run-time dependence checking. If a dependence violation is detected, the violating thread is rolled back and subsequent threads squashed. It is possible to set a maximum number of active threads (number of processing elements) or to let every available thread run simultaneously.

As opposed to a real machine, only instructions of importance for the simulation are supported; i.e. module calls, loads and stores, returns (including the actual return value) and an instruction marking that the return value was used. These events are included in the traces. A virtual timer, which keeps track of the number of instructions executed between such events, is associated with each thread.

Traces with all information needed by the analysis tool were obtained by running the programs sequentially on a system-level instruction set simulator, Simics [8]. Simics makes it possible to run applications and OS in a simulated environment, and to capture memory accesses and register contents without introducing any overhead in the application. Another feature of Simics we use is to annotate the programs to make a call-out from the application to the memory system simulator to mark an event in the program in the same way as any memory system event. This feature was used to mark module calls and returns, as well as the first occurrence of return value use.

The simulated processor is a single-issue in-order SPARC v8. The memory system is assumed to be perfect, loads and stores are always available for use in the next clock cycle. This means that the simulated system always completes one instruction each cycle.

Realistic processor core and memory hierarchy models would affect the run-time of each module: ILP would decrease thread execution time on a modern superscalar core, and an imperfect memory hierarchy would increase execution time. There are many issues affected by the memory hierarchy, for instance the impact of inter-thread communication, speculative state management, context switch overhead, sharing overhead if separate caches are used, and increased bandwidth requirements because of speculation. To determine exactly how this would affect speedup, a cycle-accurate simulator of a CMP with speculation support is needed. For the time being, we have omitted these points of the design space in favor of what we consider to be more fundamental issues. Processor and memory hierarchy considerations are very important, however, and an interesting topic of future papers.

The programs were compiled with the GNU Compiler Collection (GCC) 2.95.2 with full optimizations. In a Java Virtual Machine (JVM) environment, the execution of a Java program includes class loading and verification, Just-In-Time (JIT) compilation and/or interpretation, and garbage collection. In our measurements, the Java programs are run without a JVM. Instead, they are compiled to native executables, which means neither class loading and verification, nor interpretation or JIT-compilation occurs. Furthermore, garbage collection has been disabled. Since our intention is to find the parallelism inherent in the applications, and not to evaluate the Java run-time system, this method makes sure our measurements only contain code execution. In addition, it gives a fair comparison between Java (Object-Oriented) and C (Imperative) codes, since GCC can be used to compile all of the benchmarks. It should be noted, however, that the Java compiler is still under development, and optimizations are not as good as for the C compiler, so in comparison, the Java program instruction counts might be somewhat higher than what would be achieved with a production-quality compiler.

Only modules in the actual application are marked by the compiler. This means we do not speculate on library (or class library for Java) calls. Such functions are run inside the caller thread. Another noteworthy detail is that exceptions and I/O operations would inhibit the ability to run threads speculatively in a real machine. These events are rare in our benchmarks, so they have not been considered in the simulations. Artificial dependences through the stack from the sequential execution have been removed.

Figure 2 summarizes the simulation process: First the application is compiled with GCC, and annotations to make call-outs to Simics are inserted; then it is run on top of Simics, generating the trace; and finally the trace is 'executed' on the architectural models that collect the statistics we will present in the subsequent section.

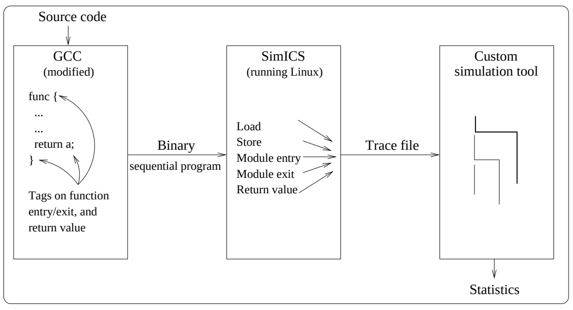

Figure 2. Our toolchain

3.2. The benchmarks

We have selected ten benchmarks, four written in C (imperative) and six written in Java (object-oriented). The C benchmarks are from the well-known SPECint95 benchmark suite and have been used in earlier STLP limit studies [12, 9]. From the eight SPECint95 benchmarks, we chose four that based on the earlier studies seem to represent typical behavior.

Three of the Java benchmarks are from SPEC JVM98. Unfortunately, the rest of the benchmarks in the suite did not include source code. Instead, we included two benchmarks from jBYTEmark (also used in [2]) and a constraint solver benchmark.

In order to keep simulation times down, we had to restrict the size of the input data sets. While the data sets are small, there are still plenty of module calls to speculate on. We have tried to make sure this restriction does not affect the behavior of the programs (for instance resulting in large initialization phases); however, it cannot be ruled out that larger input sets could affect the result on some of the benchmarks.

Table 1 briefly explains what each benchmark does. It also includes dynamic instruction and module counts, as well as average module size, for the applications. However, it should be noted that the average module size can be a bit misleading; module sizes vary greatly. In Section 4.4 we will see that that a majority of modules are less than 100 instructions in all but two of the programs.

4. Experimental results

In this section, we present the results of our experiments on the set of models described in Section 2.2 using the methodology in Section 3. We begin with studying the upper bound on the module-level parallelism in Section 4.1 followed by the impact of data dependences in Section 4.2, impact of limited processing resources in Section 4.3, and finally impact of thread-management overhead in Section 4.4.

4.1. Limits on the inherent parallelism

Figure 3 shows the speedup for our benchmark applications with perfect (i.e. always correct) value prediction both for return values and all memory loads. The harmonic mean for both groups (C and Java) of applications is also included. The speedup under ideal machine conditions is only limited by the module-level parallelism inherent in the program structure as constrained by the control dependences, i.e., how often and when modules are called. Figure 3 therefore serves as a fundamental limit for MLP, given the simplistic execution model where we begin speculation whenever a module call is encountered. The only way to find more MLP would be to speculate on module calls, i.e. speculatively call modules before execution has reached the point of the call, which introduces the element of control-speculation. To some extent, it could also be possible to use compiler transformations to rearrange the module calls in a more advantageous way, i.e. to increase the overlap of module execution.

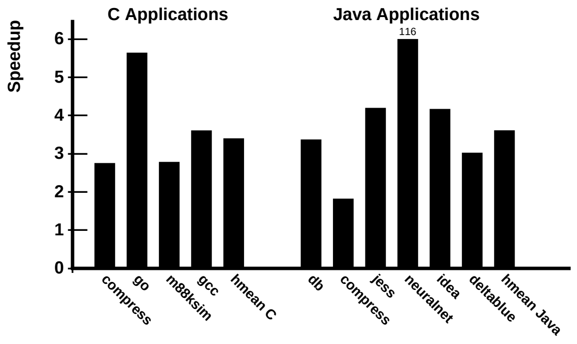

Figure 3. Speedup on the ideal machine with perfect memory and return value prediction

A noticeable fact is that the speedup without the impact of dependences is not spectacular, with a harmonic mean of 3.4 and 3.6 for the C and Java applications, respectively. This means that the module calls are not arranged in such a way that there will be a large overlap of modules even if all calls are parallelized. A contributing reason as to why we do not get a large additive effect is that the majority of modules are small. However, if some of the parallelism can be extracted with reasonable effort, it might still be a useful proposition given the simplicity by which this parallelism can be extracted from existing programs.

We can also see that there is no significant difference between the Java and C application with respect to potential MLP, the mean speedup is almost exactly the same for both

Table 1. The benchmark applications

Name	*Origin*	*Description*	*#Instructions (dynamic)*	*#Modules (dynamic)*	*Avg. instr./mod. (dynamic)*
C Applications					
compress	SPECint95	Unix compress	1.4M	21k	67
go	SPECint95	Plays the game of Go	1.4M	1.1k	1190
m88ksim	SPECint95	A chip simulator	2.2M	0.5k	4767
gcc	SPECint95	GNU C Compiler 2.5.3	13M	54.5k	237
Java Applications					
db	SPEC JVM98	Simple database	13M	4.9k	2644
compress	SPEC JVM98	Compress (Java port)	2.7M	31.5k	84
jess	SPEC JVM98	Expert system	16.3M	25.8k	633
neuralnet	jBYTEmark	Neural network	4.2M	2.6k	1626
idea	jBYTEmark	En/decryption	35.7M	12k	2966
deltablue	Sun Labs	Constraint solver	2.6M	12.5k	208

programming styles.

The reason for the high speedup (116) in NeuralNet is that a number of modules are called repeatedly inside a main loop, encompassing the entire program except for a short initialization phase. Thus, the main loop uncovers large amounts of MLP.

4.2. Impact of data dependences

The previous model predicted speedup under the assumption that value prediction on return as well as memory values is perfect. Perfect value prediction is of course not possible to attain, so the first question on our trek towards a realistic machine model is: how would value predictors with reasonable implementation complexity affect speedup?

In Figure 4, we show how memory load value prediction (MVP) affects performance. For each application the left bar, labeled (P), is the speedup with perfect MVP, while the right bar, labeled (N), indicates speedup with no MVP. The difference in height thus indicates the potential of memory load value prediction. The two rightmost bars once again show the harmonic mean.

The lack of memory value prediction has a substantial impact on some of the benchmarks. For instance, most of the massive potential in the NeuralNet benchmark disappears. NeuralNet contains numerous shared data structures that are continuously updated in each iteration of a main loop; therefore, this main loop is not possible to parallelize. The remaining parallelism comes from partial overlap of modules within a loop iteration. The key methods in NeuralNet do contain a good amount of loop-level parallelism, which cannot be exploited with the MLP-only approach. In [2], the authors have extracted module-level parallelism from this application by recoding it, converting loop-level parallelism to MLP.

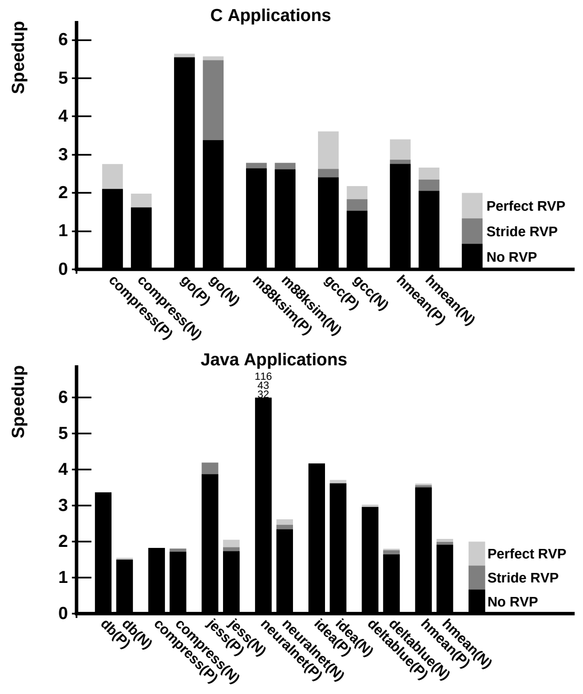

Figure 4. Value prediction: the left bar (P) for each application has perfect memory value prediction, the right bar (N) has no memory value prediction

The shaded vertical sections on each bar in Figure 4 show the impact of return value prediction (RVP). Three policies are presented: no RVP, stride RVP, and perfect RVP. For the no RVP policy, modules are still run speculatively, but a rollback always occurs to the point where the return value is used. The gap between no RVP and perfect RVP will reveal the potential benefits of return value prediction. We also included a known and computationally simple value predictor as an indication of the predictability of return values; the stride predictor, which predicts the next value as the last value plus the difference between the two last values. Another obvious candidate would be a last-value predictor, however, they both catch the most obvious case of a function that almost always returns the same value.

Return value prediction seems to make sense in some of the benchmarks, but surprisingly, in many of the programs most of the parallelism can be exploited without RVP, since a large portion of the modules either do not produce a return value at all (void modules), or produces a return value which is never used. If one would choose a scheme without RVP, a speculation system could catch and rollback modules in the cases where the return value is indeed used, but a better way would be to have the compiler mark all calls to void modules and those whose return value is not used, since this can be determined statically.

The simple stride value predictor has been observed to perform reasonably well in most applications, successfully predicting between 20% and 80% of return values in seven of the ten applications. Some applications, notably Idea and NeuralNet, did show a very large percentage of mispredictions. It turned out to be because of heavy use of a random number function during initialization (which is a small part of the total execution time). It would be useful to be able to selectively disable speculation for such cases, where obviously no value predictor can be expected to perform well.

The execution time in Idea is concentrated to one module which handles encryption/decryption. Most of the speedup we can see for this program is because of overlap of two iterations of encryption and decryption (four calls to this module). Although it is written in Java, it has the structure of an imperative program, which is not surprising considering the fact that it is converted from C. NeuralNet and Java Compress are also originally C programs.

Our initial belief was that the object-oriented (Java) programs would exhibit more parallelism than the imperative (C) at this point for two reasons: the object-oriented programming style encourages more frequent use of module calls, and the use of data encapsulation would result in fewer memory dependences. However, our results do not indicate any such difference (Figure 4). There seems to be a small difference when it comes to return values, the speedup of the C programs are slightly more affected by rollbacks due to return value mispredictions.

In summary, two important lessons can be learned from this experiment: a simple return value predictor will suffice in most cases, and a good memory load predictor would be very useful. In the rest of this paper, we will assume no value prediction on memory loads, and stride value prediction for return values, since we feel that this represents a design choice of reasonable complexity today.

4.3. Impact of limited processing resources

In Figure 5 we can see the speedup for our applications running on a machine with limited processing resources. The model is still ideal in the sense that we assume the processing elements (PEs) on an *n*-way machine can always be utilized executing the *n* least speculative threads. A more speculative thread will be preempted, without penalty, if a new less speculative thread arrives; execution will be resumed, however, where it was preempted the next time the thread can be rescheduled on a PE.

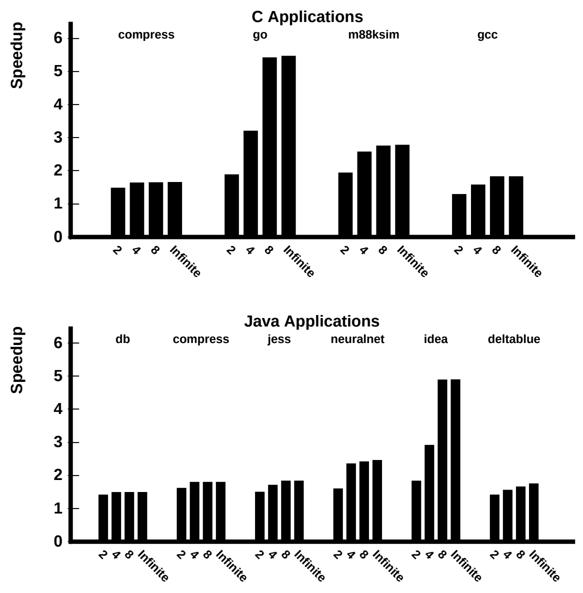

Figure 5. Speedup with 2, 4, 8 or an infinite number of PEs

With this model, we can see that virtually all potential speedup can be utilized with only eight PEs. In fact, many of the benchmarks do not benefit significantly from more than four PEs. This is good news, since it shows that parallelism is in general not concentrated to a limited part of the execution; rather, a limited number of PEs are busy working most of the time.

This data suggests that all the module-level parallelism available in C and Java programs could potentially be exploited using chip multiprocessors with relatively few processor cores. Again, there is not any big difference across C and Java programs.

4.4. Impact of thread-management overhead

Figure 6 shows speedup with overhead for speculation support. We have included three types of overhead: starting a new speculative thread, performing a rollback on misspeculation, and committing speculative state when a thread has successfully finished. A thread that has been squashed as part of a rollback will incur a new thread start overhead when it is called again.

In the figure, the three types of overhead are set to the same size; we ran simulations for 10, 100, or 1000 cycles. For the sake of comparison, we repeat the speedup for the no-overhead machine. The numbers are for an 8-way machine.

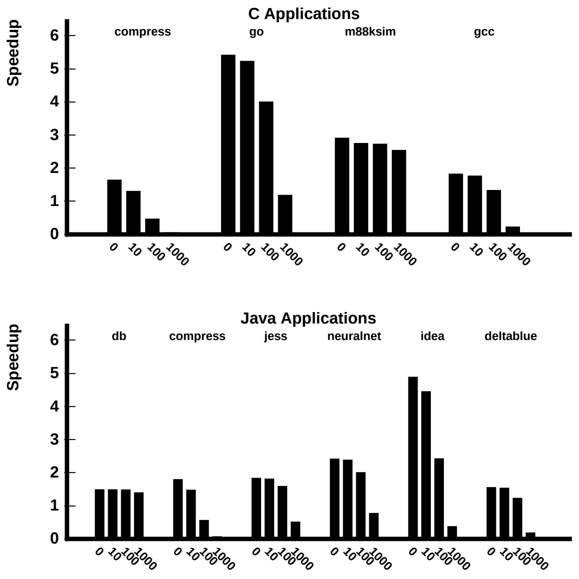

Figure 6. Speedup with thread-management overheads of 0, 10, 100 or 1000 cycles on an 8-way machine

The 100-cycle overhead simulations are interesting since they approximately correspond to the overheads reported for module speculation support in the Hydra CMP [6]. At a 100-cycle overhead, we can already see a severe impact on the speedup for several applications, even a slowdown for both compress programs. When the overhead is increased to 1000 cycles, the compress programs are more than ten times slower than their sequential execution.

In order to find the reason for this, we do not need to look further than module size. Figure 7 reveals that for both C and Java compress, the majority of the modules are shorter than 20 cycles, and almost all are under 100 cycles. This means that thread-management overheads will dominate execution time, since each module will, at least, give rise to a thread start overhead when called, and a commit when it reaches return. On the other hand, some of the modules are very large, which explains why the average sizes presented in Table 1 are several thousand instructions for some of the programs. Note that with our single-issue, perfect memory machine, there is a one-to-one correspondence between the number of cycles and instructions.

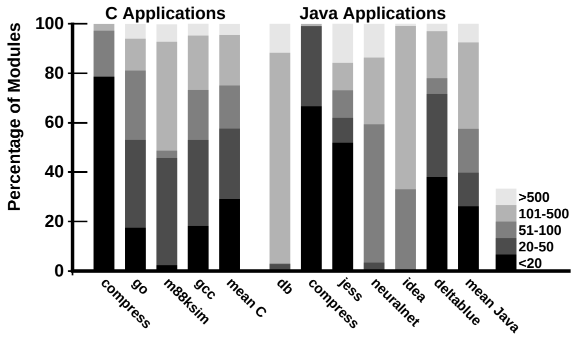

Figure 7. Percentage of modules (dynamic) of size <20, 20-50, 51-100, 101-500, or >500 cycles

We have observed that one of the side effects of increasing the number of processing elements is that the number of dependence violations will also increase. Therefore, with high thread overheads, the benefits of adding more processing elements will be smaller than indicated in Figure 5, in some cases we can even get a slowdown.

In Figure 8 we can see how the execution time is used. The execution time for a program in this figure is the total used time on all PEs added together. The simulations are run with 100-cycle thread-management overheads on eight PEs.

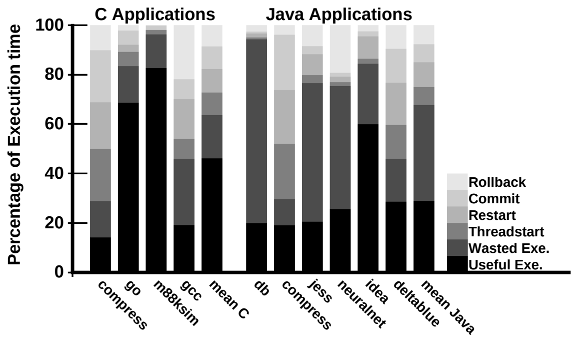

Figure 8. Part of execution time that is useful, wasted because of rollbacks, and used for thread-management. 100-cycle overheads on an 8-way machine

Useful execution is the part of the execution that was successful and committed; it is the part that corresponds to the sequential execution. Wasted time is the execution time that was thrown away because of a rollback or when the thread was squashed. Restart is the effect of additional thread start overhead for a thread that was squashed and must be started again. The remaining three categories show thread start, rollback, and commit overhead.

This figure points out one of the serious disadvantages of speculative execution. Only 20% on average for Java programs, or 40% for C programs, of the processing time is useful execution. This is a disadvantage in a multitasking environment where other processes might make better use of the resources. It is also a problem from an energy-efficiency perspective. Wasted execution makes up a major part of the total processing time for most benchmarks. A conclusion would be that methods for minimizing the number of misspeculations, and thus wasted execution, is probably needed even if it is not necessary from the point of view of performance for a single-application.

It is clear from this experiment, however, that keeping overheads small is of utmost importance for module speculation support.

5. Related work

There is a large body of research in the recent literature that focuses on architectures and compilation techniques for speculative thread-level parallelism.

One of the first architecture proposals for thread-level speculation was done within the Multiscalar project [15]. One of the novel features of this architecture is the address-resolution buffer [4] that validates and signals violations to data dependences between threads. Another noticeable speculative architecture proposal is the superthreaded architecture [17]. Several distributed approaches to implement support for thread-level speculation have also been presented in the framework of chip-multiprocessors [5, 6, 16]. This study is based on the feasible inclusion of such a mechanism in chip multiprocessors.

Another important prerequisite for this study is the progress in value prediction done over the last few years. Value prediction enables speculation beyond the data flow limit. It was introduced by Lipasti *et al.* [7] as a way to hide memory load latency by allowing data dependent instructions to execute in parallel. The predictability of data values was investigated in [14]. Others have followed up with a number of inventive prediction schemes such as the stride and last value predictors [7] which we study in this paper.

This paper focuses on the opportunities and limitations of speculative module-level parallelism – a straight-forward method to extract thread-level parallelism out of existing software. Several recent papers have had similar goals. A limit study of the inherent loop-level as well as module-level parallelism in SPECint95 programs was recently published by Oplinger *et al.* [12]. While disregarding architectural limitations in terms of thread management overheads, they found that there is ample module-level parallelism in the benchmark suite that can be exploited by multiprocessor or multithreaded processor cores of typically less than eight processors. In comparison with our study, they didn't address how important memory-level dependences are and did not look at Java applications. Moreover, they didn't study how much typical overheads in CMP architecture models would affect the achievable speedup.

In contrast, Chen and Olukotun [2] focus on Java programs. Their study is mostly aimed at the speedup obtained on the Hydra CMP proposal and does neither address the impact of various value prediction schemes nor how scalable the performance is. While they note that thread management overhead may have a severe impact on the speedup, they didn't analyze how it relates to the size of the modules. In a follow-up study by the same team based on SPECint95 programs [11], they observe that thread-management overheads can be detrimental to the speedup obtained because of the penalties associated with misspeculations. As a remedy, they propose and evaluate schemes that select modules to speculate on depending on their likelihood to succeed.

Value prediction as a way to reduce dependence violations in thread-level data dependence speculation architectures has been investigated by Marcuello *et al.* [10, 9] in the context of their Clustered Speculative Multithreaded processor. They speculate on live input values to threads (values used but not defined within the thread) at thread start time. Some works mentioned earlier has also used value prediction for module return values [2, 6] and memory loads [12]. Others who have used value prediction in conjunction with coarse-grained speculative architectures include [13, 1, 3]. However, to the best of our knowledge, our study is the first to address the limits on value prediction which pin-points whether there is room for improvements.

6. Conclusions

The goal of this study has been to understand the impact of the programming style – imperative versus object-oriented – on the inherent module-level speculative parallelism as well as how architectural deficiencies in proposed chip-multiprocessor architectures affect the achievable speedup.

One would expect that object-oriented programs would make more heavy use of modules and would encapsulate many of the data dependences with a potential to expose more module-level parallelism. Contrary to this intuition, we found that there is not any significant difference between the inherent module-level parallelism in the C versus the Java programs that we studied. In both cases, we observed a speedup limit of about 3.5. In addition, the two suites representing the two programming styles were both sensitive to memory-level data dependences which suggests that progress in memory value prediction schemes are important to approach the maximum speedup. As for return-value prediction schemes, simple ones based on last- or stride-value fare pretty well across all applications.

When considering the impact of architecture-level constraints, we found that all of the inherent parallelism could be exploited by typically small multithreaded or multiprocessor cores with less than eight processors. However, a key inhibitor to reaching the speedup limit is the overheads imposed by thread management including the time to start (or restart), commit, or roll-back threads upon data dependence violations. Given the fairly small module sizes, speedup is severely affected when the overhead exceeds a hundred cycles. This calls for more efficient thread management mechanisms than what is currently known in the literature for chip-multiprocessor architectures. Obviously, using MLP in more loosely coupled architectures is not an option.

In this study, we did not try to enforce a certain thread granularity, instead all modules were parallelized regardless of size. On realistic architectures, with various overheads, granularity is important. Methods to selectively apply module-level speculation would be needed. Our reason for using modules as the only source of parallelism was that they are control independent and easy to identify. Since the amount of MLP is limited, additional sources of parallelism are needed in order to achieve large performance gains using thread-level speculation techniques, but likely at the expense of increased complexity.

Acknowledgments

We would like to thank Peter Rundberg and Magnus Ekman of Chalmers University of Technology, as well as the anonymous reviewers, for their valuable comments on earlier drafts of this paper. This research has been financially supported by the ARTES/PAMP program of the Swedish Foundation for Strategic Research (SSF) and by equipment grants from Sun Microsystems Inc. and the Swedish Council for Planning and Coordination of Research (FRN) under contract number 96238.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 226–236. IEEE Computer Society, Dec. 1998.

[2] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 176–184. IEEE Computer Society, Oct. 1998.

[3] L. Codrescu and D. S. Wills. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 International Conference on Computer Design (ICCD '99)*, pages 428–435. IEEE Computer Society, Oct. 1999.

[4] M. Franklin and G. Sohi. Arb: A hardware mechanism for dynamic memory disambiguation. In *IEEE Transactions on Computers Vol. 45 No. 5*, pages 552–571. IEEE Computer Society, May 1996.

[5] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 195–206. IEEE Computer Society, Feb. 1998.

[6] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII '98)*, pages 58–69. ACM Press, Oct. 1998.

[7] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII '96)*, pages 138–147. ACM Press, Oct. 1996.

[8] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 119–130. USENIX Association, June 1998.

[9] P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 595–604. IEEE Computer Society, May 2000.

[10] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings. 32nd Annual International Symposium on Microarchitecture (MICRO '99)*, pages 230–237. IEEE Computer Society, Dec. 1999.

[11] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 21–30. ACM Press, June 1999.

[12] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 303–313. IEEE Computer Society, Oct. 1999.

[13] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 138–148. IEEE Computer Society, Dec. 1997.

[14] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 248–258. IEEE Computer Society, Dec. 1997.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425. ACM Press, June 1995.

[16] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, Feb. 1998.

[17] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46. IEEE Computer Society, Oct. 1996.