Limits on Thread-Level Speculative Parallelism in Embedded Applications

Mafijul Md. Islam[1], Alexander Busck[1], Mikael Engbom[1], Simji Lee[2], Michel Dubois[2], Per Stenström[1]

[1]Department of Computer Science and Engineering,
Chalmers University of Technology,
SE-412 96 Göteborg, Sweden

[2]Department of Electrical-Engineering Systems,
University of Southern California,
Los Angeles, USA

Contact Person: Mafijul Md. Islam, **Email:** mafijul.islam@ce.chalmers.se

ABSTRACT

As multi-core microprocessors are becoming widely adopted, the need to extract thread-level parallelism from sequential single-threaded applications in a seamless fashion increases. In this paper, we study the limits of performance speedup for embedded applications using parallelizing compilers on platforms with and without support for thread-level speculation.

First and somewhat expected, only two out of ten applications from the consumer and telecom domains of the EEMBC suite could be automatically parallelized on multi-core architectures with no thread-level speculation (TLS) support. We systematically study the speedup obtained by parallelizing compiler technologies by factoring in the impact of the number of cores, thread decomposition strategies, and thread-management overhead. Overall, we have found that a TLS substrate is critical to uncover thread level parallelism and thread-management overhead must be low. On an eight-way multi-core system, it is possible to achieve a speedup of four, on average, for six out of the ten applications of EEMBC which we have analyzed.

1. Introduction

Frequency-scaling and exploitation of instruction-level parallelism have been the two major contributors to the performance doubling every eighteen months as predicted by Moore's law. Unfortunately, while both sources are far from exhausted, it has become challenging to continue leveraging them. For the desktop and server segments, heat dissipation on the microprocessor die has made frequency-scaling less attractive. And, the increasing wiring delays together with exponentially growing reorder buffers have made further exploitation of instruction-level parallelism practically infeasible [1]. As a result, virtually all microprocessor vendors have now launched *multi-core* roadmaps in which the doubling of transistors every eighteen months is likely to double the number of cores on die.

In the embedded segment, multi-cores are also expected to become an important technology. For feature phones which have a rich functionality in terms of multimedia support, the inherent thread-level parallelism can potentially be uncovered by multi-core microprocessors. Moreover, dynamic-voltage-frequency-scaling methodologies can be used to reduce the power consumption to make the battery last longer.

Most code is however sequential and single-threaded. Despite research into parallelizing compilers for many decades, few codes can be automatically parallelized. This is because of the lack of information and/or limitations in static pointer analysis frameworks to chase all dependences at compile-time. This results in conservative decisions that make most available thread-level parallelism unexploited. As a remedy, there has been a significant amount of research into *thread-level speculation* (TLS) (see, e.g, [9, 10, 11, 13, 14, 15, 16]), which postpones the dependency check to run-time. Assuming that multi-core chips offer a TLS substrate, the compiler can more aggressively expose thread-level parallelism.

This is the first study to establish the amount of thread-level parallelism that can be uncovered in the entire consumer and telecom suites of EEMBC [4] using a parallelizing compiler framework assuming a multi-core model with and without support for TLS. We systematically study the speedup obtained by parallelizing compiler technologies by factoring in the impact of the number of cores, thread decomposition strategy, and thread-management overhead. Overall, we have found that a TLS substrate is critical in uncovering thread-level parallelism and thread-management overhead must be low. On an eight-way multi-core system, it is possible to achieve a speedup of four on average for six out of the ten applications in EEMBC which we have analyzed.

In the next section, we provide the details of the thread-level speculative model that we use. In Section 3, we provide details of the applications used along with the experimental strategy. Characterizations of loops are done in Section 4. Experimental results are presented in Section 5 followed by a discussion of related work in Section 6 before we conclude in Section 7.

2. Execution and Architectural Models

The goal of this study is to establish how much speedup can be obtained using a parallelizing compiler with and without support for TLS for a suite of embedded applications. In this section, we first present the software execution model used to extract TLP in Section 2.1 and then the suite of architecture models used in our experiments to establish the amount of parallelism that can be uncovered. This is done in Section 2.2.

2.1 Speculative Loop-level Execution Model

Our model for extracting threads from single-threaded programs is based on speculative loop-level parallelism. Successive loop iterations are speculatively run in parallel. Figure 1 shows the loop-level execution model. The program to the left in Figure 1 contains an example loop with a trip count of *m*. Assuming $m = 3$, the sequential execution time and the execution time with TLS support when the iterations are independent are shown to the right in Figure 1.

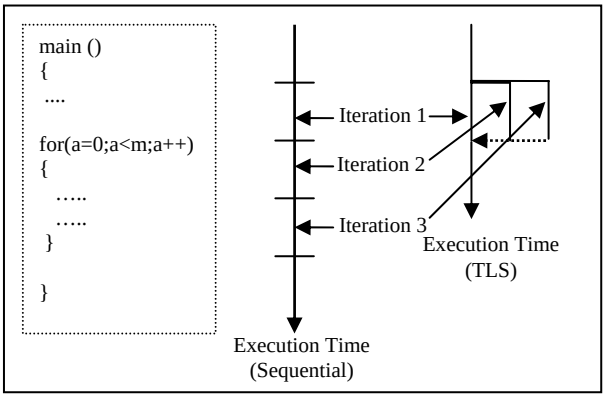

Figure 1: Speculative loop-level execution model

2.2 Architectural Models

We initially assume an idealized multi-core architecture with as many cores as the number of iterations in a loop. Moreover, we do not factor in any performance losses either in the memory system or due to thread-management. When a speculative loop is encountered, the same number of threads as the number of iterations is spawned. If a thread with a higher iteration number has speculatively read from a memory location that a thread with a lower iteration number has modified, and this modification *happens after* the read operation, the former thread along with all threads associated with subsequent iteration numbers are squashed. Once all successfully spawned speculative threads have terminated, the squashed threads are then speculatively spawned using the same procedure until all iterations have been executed. We have opted for this simple model of re-execution but note that one could have spawned the squashed threads immediately after the violation. It is an open question which of the strategies would be best.

In principle, the execution time of the speculatively executed loop is dictated by the depth of the dataflow graph where each node is a single iteration. The algorithm for determining the execution time of the speculative loop is shown in pseudo-code in Figure 2.

```
time=0;
first_iter=1;
for (i=1; i <= no_of_iter; i++)  {
   my_iter = i;
   for (j=i-1; j >=f irst_iter; j--) {
     if  (my_iter  mod (P+1)  !=  0)  then {
       if  is_dependent(my_iter,j) then {
          time=time + max_time(first_iter, my_iter-1) +
                thread_management;
          first_iter = my_iter;
        }
     }
     else {
       time = time + max_time(first_iter, my_iter) +
              thread_management;
       first_iter = my_iter;
     }
   }
}
```

Figure 2: Pseudo-code for determining the execution time for a speculative loop.

The outer loop visits iterations one by one. For each iteration (denoted my_iter), the first task is to determine whether there is a dependency with a previous iteration by visiting all iterations which were executed concurrently. The function is_dependent checks dependences between two iterations. If there is a dependency, time is elapsed by an amount that corresponds to the time for the longest iteration, since all concurrent iterations must be terminated before the next group of speculatively executed iterations can start.

When modeling a multi-core system with as many cores as the number of iterations, P is initialized to the number of iterations denoted no_of_iter. Then, the condition of the first if-statement will never be satisfied. On the other hand, when modeling a multi-core system with a limited a number of cores (P), this

same condition is satisfied once every P iterations and time is then advanced. This is what is taken care of by statements in the else clause of Figure 2.

So far we have assumed that thread management incurs no overhead. To factor in thread-management overhead in the execution time of a speculative loop, we charge thread_management cycles every time a new group of P speculatively executed iterations is launched. The extent to which the thread-management overhead impacts on the execution time is dictated by the size of each thread. Because a single iteration, so far, corresponds to a thread, the number of instructions executed per iteration is important. We will also experiment with other thread decomposition strategies to cut down on the impact of thread-management overhead. Given N iterations and P cores, one natural decomposition strategy is to assign the first N/P iterations to the first core, the second N/P iterations to the second core, and so on. This coarse grain decomposition will make each thread bigger and thread-management overhead will have a relatively lower impact on the execution time.

3. Methodology

We start this section with a brief description of the benchmarks used in this study. Then, we present a detailed description of the simulation framework.

3.1 Benchmarks

We have used the ten applications from the telecom and consumer domains of the industry-standard embedded benchmark suite EEMBC 1.1 [4]. All of them are written in C. The benchmarks were compiled to the MIPS-like PISA architecture using the GCC Cross Compiler of the SimpleScalar tool set assuming optimization level -O3 [3]. The numbers of distinct loops, that is, the *static loop count* and the dynamic instruction count for each application have been obtained by running the applications using *sim-fast* of the SimpleScalar tool set [3]. This study has considered only the loops present in the application code and ignored the loops which are part of the library code. Table 1 presents the applications used in this study. It is evident from Table 1 that most of the applications, except cjpeg and djpeg, contain a small number of loops.

3.2 Parallelizing Compiler

We have used the *Intel® C++ 9.1 Compiler for Linux* [6] to determine the extent to which a parallelizing compiler can uncover thread-level parallelism in the embedded applications. The Intel® compiler includes advanced optimization features such as *Auto-Parallelization* which improves application performance on multiprocessor systems by means of automatic threading of loops [6].

Table 1: Number of static loops and the dynamic instruction count of the EEMBC applications.

Benchmark	Description	Dyn. Instruc. Count (MInst)	Loop Count (Static)
autocor00	Telecom, Fixed Point Autocorrelation	7	10
conven00	Telecom, Convolutional Encoder	666	8
fbital00	Telecom, Fixed Point Bit Allocation	2240	9
fft00	Telecom, Fixed Point Complex FFT/IFFT	54	14
viterb00	Telecom, Viterbi Decoder	893	13
cjpeg	Consumer, Jpeg Compression	20960	100
djpeg	Consumer, Jpeg Decompression	17081	108
rgbcmy01	Consumer, Image Filter	2523	11
rgbhpg01	Consumer, Image Filter	406	7
rgbyiq01	Consumer, Image Filter	696	10

This option detects parallel loops capable of being executed safely in , automatically generates multi-threaded code, and relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling, and synchronizations [6]. We have used the optimization level –O3 and some additional flags - parallel and par-report - to compile and auto-parallelize the applications. The *parallel* flag is used to detect the loops which can be executed safely in parallel. The *par-report* flag is used to generate the detailed report containing information such as how a particular loop can be parallelized and why a particular loop cannot be parallelized. The results obtained are presented in Section 5.1.

3.3 Simulation Framework

All the results presented in this paper are obtained from our custom trace-driven simulation tool. Figure 3 summarizes the simulation process. Firstly, as mentioned in Section 3.1, we have compiled the applications using the cross-compiler of SimpleScalar

tool set [3]. Secondly, the generated binaries are fed into *objdump* tool of GNU binutils of SimpleScalar [3] as input to find the correspondence between the source code and the assembly code of each of the applications. We have used the output of *objdump* to determine the start PC and the end PC of each loop as well as the start PC of library code.

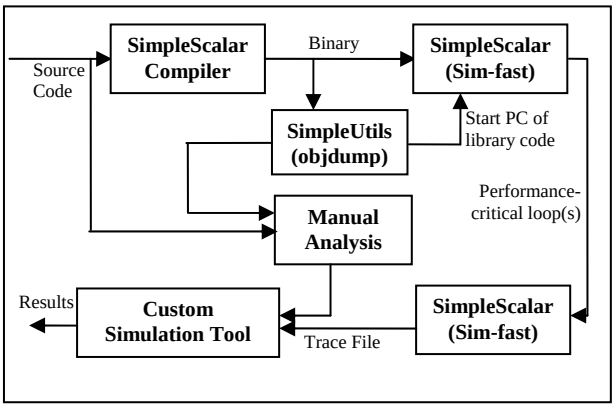

Figure 3: Simulation Framework used in this study

Thirdly, we have modified *sim-fast*, a fast functional simulator of SimpleScalar tool set [3], to implement the dynamic loop detection method described in [11] to calculate the execution-time coverage of a loop. We have executed the programs using the modified version of *sim-fast* to which the start PC of library code has been provided as one of the arguments. For each of the loops present in the application code, we have identified start PC and end PC of the loop, its execution-time coverage and whether it is the innermost loop (if nested). Note that we only attempt to parallelize the innermost loop in a loop nest.

We then counted the total number of dynamic instructions in a particular loop as well as in the application. The dynamic instruction count of the loop is divided by the dynamic instruction count of the application to calculate the *execution-time coverage* of the particular loop. The output of this stage is a set of performance-critical loops – loops with execution-time coverage higher than a pre-set threshold – for each of the applications.

At the next step, the output of *objdump* is used to find the exact location of each of the performance-critical loops in the source code through manual analysis. Then, we have performed manual analysis of each of the loop body to identify the variables/statements to which privatization and parallel reduction may be applied to exploit and enhance parallelism. Then, we have generated a separate trace file for each of the performance-critical loops by simulating the programs using another modified version of *sim-fast* and the knowledge gathered through manual analysis.

We simulate a single-issue, in-order processor running Linux which completes one instruction each cycle. We have assumed a perfect memory system in which loads and stores can always be used in the next clock cycle. Though considerations of more realistic processor and memory hierarchy models are very important, right now we are only interested in establishing an upper bound on the speedup obtained using speculative loop level parallelism in the telecom and consumer suites of EEMBC. The discovered potential may later be realized/exploited using more realistic models. Finally, the generated trace files are used as input to the custom simulation tool that implements the algorithm determining the execution time according to Figure 2.

4. Loop Characterization

In this section, we present detailed results for the performance-critical loops of the selected applications of EEMBC 1.1.

We have determined not only the location of the loop in the source code but also its execution-time coverage using the methodology presented in Section 3. The loop trip and the dynamic number of instructions per iteration are important because they serve as a guide to decide on whether the loop should be parallelized and the maximum number of threads to be spawned simultaneously considering the available resources. Firstly, the iterations with a low instruction count are supposed to be more sensitive to thread-management overhead than those with a high instruction count. Secondly, variations in the instruction count among the iterations may lead to load imbalance across the processors. So we have determined the loop trip count as well as the average number of instructions per iteration for the selected loops. The results are shown in Table 2.

It is noticeable from Table 2 that there is often a single performance-dominating innermost loop which could potentially yield significant speedup if it has no loop-carried dependences. In fact, all applications except **cjpeg** and **djpeg** have such a performance-critical loop. For example, **autocor00** has a loop invoked 40000 times with 85% execution-time coverage and performance-critical loops, e.g., the loops in **fbital00, rgbcmy01, rgbhpg01** and **rgbyiq01**, have high loop trip counts of 256, 4161, 318, and 4714, respectively. This implies that these applications might show substantial speedup. There are counter examples as well – **conven00** has a loop trip count of 5 for the loop with 60% execution-time coverage and we would

expect that parallelization of the loop might not yield reasonable speedup for this application.

In general, we observe that the number of instructions in each iteration is low (<100) in most of the loops which implies that thread-management overhead is expected to have a significant impact on the achievable speedup. For example, the average number of instructions per iteration is 13 in **autocor00** whereas the number is 86 in case of **rgbyiq01**. Thus when parallelized, we may expect **rgbyiq01** to perform better compared to **autocor00** with an increase in thread-management overhead.

Table 2: Characteristics of performance-critical loops

Application	Location of Loop (Filename : Line No)	Execution-time Coverage	Max. No of Iterations /Invocation	Avg. No of instructions /Iteration
autocor00	autocor00.c:111	85.00%	16	13
conven00	conven00.c:130	60.00%	5	17
conven00	conven00.c:119	14.00%	4	10
conven00	conven00.c:127	13.00%	2	101
fbital00	fbital00.c:129	96.00%	256	27
fft00	fft00.c:172	72.00%	128	39
fft00	fft00.c:136	9.00%	256	20
fft00	fft00.c:191	8.00%	256	16
viterb00	viterb00.c:263	84.00%	16	48
viterb00	viterb00.c:310	5.00%	32	11
cjpeg	jccolor.c:212	14.00%	320	38
cjpeg	jfdctint.c:283	11.00%	8	155
cjpeg	jfdctint.c:218	10.50%	8	153
cjpeg	jcdctmgr.c:216	9.00%	64	17
cjpeg	fileio.c:396	8.00%	66651	8
cjpeg	rdbmp.c:551	8.00%	960	8
cjpeg	fileio.c:299	7.00%	14	8
djpeg	fileio.c:396	19.00%	6771	8
djpeg	jdcolor.c:208	15.00%	320	35
djpeg	jidctint.c:341	15.00%	8	187
djpeg	jidctint.c:235	10.00%	8	188
djpeg	jdsample.c:439	9.00%	158	22
djpeg	fileio.c:299	9.00%	4096	8
rgbcmy01	bmark_lite.c:271	93.00%	4161	32
rgbhpg01	bmark_lite.c:234	95.00%	318	52
rgbyiq01	bmark_lite.c:349	94.00%	4714	86

5. Results

In this section, we present and discuss the results of our application case study. Firstly, we present the achievable speedup using the parallelizing compiler from Intel® [6] in Section 5.1 with no support for TLS followed by the speedup assuming support for TLS on an ideal machine in Section 5.2. We then study the impact of constraining the number of cores and thread-management overhead on the potential speedup in Section 5.3 and Section 5.4, respectively. Finally, the impact of thread size on the speedup is studied in Section 5.5.

5.1 Speedup without Support for TLS

Figure 4 presents the obtained speedup for the selected applications using the parallelizing compiler as described in Section 3.2. We have assumed infinite number of processing elements and zero-cycle thread-management overhead. As expected, the performance improvement for most of the applications is quite poor.

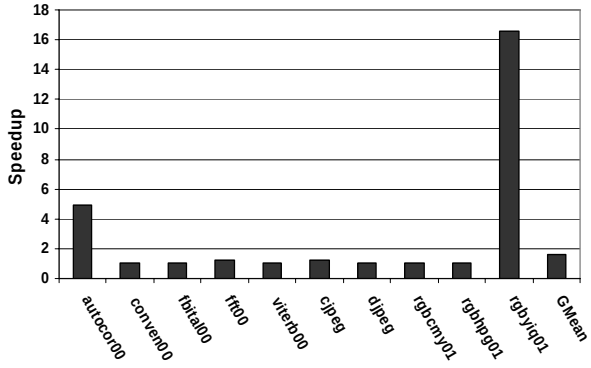

Figure 4: Speedup using Intel® Compiler

In fact, only two applications (**autocor00** and **rgbyiq01**) out of ten enjoy a significant speedup. In terms of loops, out of twenty-six loops presented in Table 2 of Section 4, only seven can be automatically parallelized by the compiler. To uncover the reason, we have analyzed the reports on parallelization generated by the Intel® Compiler for the loops which have execution-time coverage of at least 50%. The findings are shown in Table 3.

Table 3 : Auto-parallelization report

Application	Coverage	Auto-Parallelization	Remarks
autocor00	85%	Yes	
conven00	60%	No	Potential data dependency
fbital00	96%	No	Statically unknown loop trip
fft00	72%	No	Potential data dependency
viterb00	84%	No	Potential data dependency
rgbcmy01	93%	No	Potential data dependency
rgbhpg01	95%	No	Potential data dependency
rgbyiq01	94%	Yes	

We have found two main reasons why the loops cannot be parallelized automatically: 1) statically unknown loop trip and b) a *potential* data dependency. For example, the loop in **fbital00** with execution-time coverage of 96% cannot be parallelized by the compiler because of the statically unknown loop trip count.

5.2 Speedup with Support for TLS

In this section, we have assumed an ideal machine model with TLS support and an infinite number of cores. The machine model is capable of managing (spawning, committing, and making a rollback) threads in zero time. It spawns a new thread at every iteration of a loop. Dependencies due to loop variables and the variables to which privatization and parallel reduction are applicable have been taken out of the trace to obtain the results. The speedups shown in Figure 5 are for the programs and are calculated following Amdahl's Law using execution-time coverage of the selected loops.

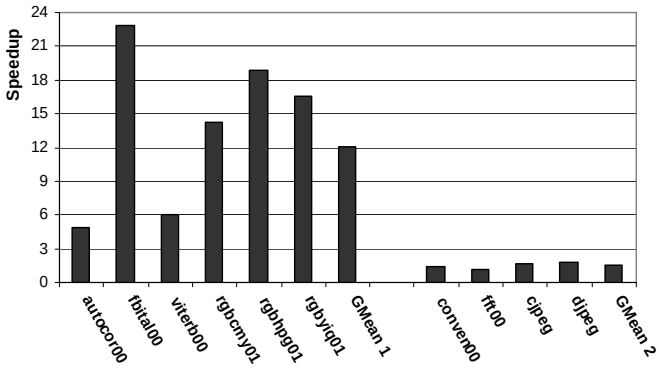

Figure 5: Speedup on the ideal machine

Surprisingly, we can see from Figure 5 that six applications (**autocor00, fbital00, viterb00, rgbcmy01, rgbhpg01** and **rgbyiq01**) out of ten yield decent speedup. We have also analyzed the other four applications (**conven00, fft00**, **cjpeg** and **djpeg**) to figure out why they do not yield better speedup. We have found that **conven00** does not show any speedup because of very low loop trip counts which are 5, 4 and 2 for the three loops with 60%, 14% and 13% execution-time coverage, respectively, as shown in Table 2. The presence of loop-carried dependences in the most performance-critical loop of **fft00** with execution-time coverage of 72% explains the reason of its lack of performance improvement.

Unlike other applications, **cjpeg** and **djpeg** do not have a single performance dominating loop. Rather, we have identified multiple loops for each of the

applications and data dependences across iterations exist in many of them which make these two applications unsuitable for exploiting thread-level parallelism. We have also analyzed the six applications which yield significant speedups to understand the variations in the obtained speedup. The loop trip count and the execution-time coverage shown in Table 2 can explain the results. For example, **fbital00** which yields the highest speedup (22.86) has a loop trip count of 256 for the loop with execution-time coverage of 96% and **autocor00** with the lowest speedup (4.92) among the first six applications has a loop trip count of only 16 and execution-time coverage is 85%.

5.3 Impact of Number of Cores

In Figure 6 we present the speedup of the applications running on a machine with a limited number of cores. We have also incorporated the results from the previous section to get a better understanding of the impact of the number of cores on speedup. All the other assumptions made in Section 5.2 are valid in this section as well. Further, an *n*-way machine will concurrently execute *n* speculative threads which are *n* successive iterations of a loop in this study.

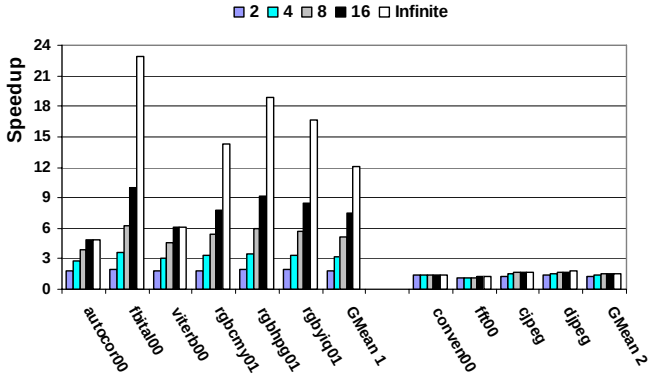

Figure 6: Speedup with 2, 4, 8, 16 or an infinite number of cores.

In Figure 6, for each of the applications we show five bars which represent the speedup on a machine with 2, 4, 8, 16 and an infinite number of cores, respectively, from left to right. We can conclude from Figure 6 that six applications (**autocor00, fbital00, viterb00, rgbcmy01, rgbhpg01,** and **rgbyiq01**) achieve linear speedup with respect to the number of cores and using more than two cores does not have any noticeable impact on the speedup of the remaining four applications (**conven00, fft00, cjpeg** and **djpeg**). Also, the potential speedup can be fully realized using sixteen processors for two applications (**autocor00** and **viterb00**) because of the loop trip (16) of the performance-critical loop of these applications.

5.4 Impact of Thread-Management Overhead

Figure 7 shows speedup for an 8-way machine which incurs thread-management overhead to support speculation. In Figure 7, for each of the applications we have four bars which represent the speedup on an 8-way machine with thread-management overheads of 0 (no overhead), 10, 20 and 50 cycles, respectively from left to right. Thread management typically incurs three types of overhead – starting a new speculative thread, performing a rollback on mis-speculation and committing speculative state when a thread has successfully finished [13]. In this study, we have used the term thread-management overhead to collectively refer to these three different types of overheads. We have run simulations for three different values of overhead – 10, 20, and 50 cycles. While 10 cycles is aggressive, multi-cores open up the possibility for new thread-spawn mechanisms in which the initial architectural state is broadcast. On the other hand, 50 cycles is more on par with what is achievable today.

Each thread consists of a single iteration of a loop. To have a better insight into the impact of overhead on the speedup, the results obtained using zero-cycle overhead has been reproduced.

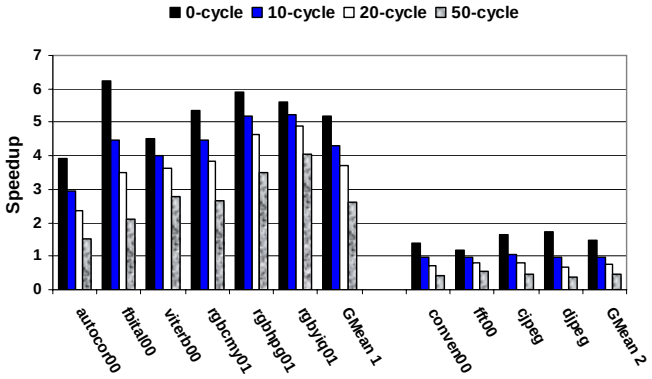

Figure 7: Speedup with thread-management overhead of 0, 10, 20 or 50 cycles.

It is evident from Figure 7 that speedup of every application is highly sensitive to the thread-management overhead. In fact, the four applications which have achieved poor speedup in Sections 5.2 and 5.3 (zero-cycle overhead) are getting slower with respect to their sequential versions for an overhead of 20 cycles or more. The remaining six applications, though achieving reasonable speedup, are getting significantly slower with the gradual increase of overhead. In order to understand this behavior, we have looked at the average number of instructions in an iteration of the selected loops as shown in Table 2 of Section 4.

We have found that the average instruction counts in **autocor00, fbital00, rgbcmy01**, **rgbhpg01,** and **rgbyiq01** are 13, 27, 32, 52 and 86, respectively. Hence, **autocor00** is expected to be the most sensitive whereas **rgbyiq01** is expected to be the least sensitive to the increasing thread-management overhead. In fact, the expectations turn out to be true if we look at the results shown in Figure 7. Moreover, there is a one-to-one correspondence between the number of cycles and instructions in our single-issue machine. Thus, it is perceivable that thread-management overhead has a significant impact on the execution time assuming a single iteration per thread. This necessitates evaluating the impact of packing multiple iterations into a single thread on speedup which we do in the next section.

5.5 Impact of Thread Size

In this section, we experiment with a coarse-grained thread decomposition strategy in which the N iterations are assigned to threads by letting each thread execute K consecutive iterations. The objective is to amortize thread-management overhead on a larger code chunk. A thread size (K) of 1, 2 and 4 iterations is considered and the corresponding results for an eight-way machine with thread-management overhead of 10 cycles are shown in Figure 8. In Figure 8, for each of the applications we show three bars which represent the speedup for a thread size of 1, 2 and 4 iteration(s), respectively, from left to right.

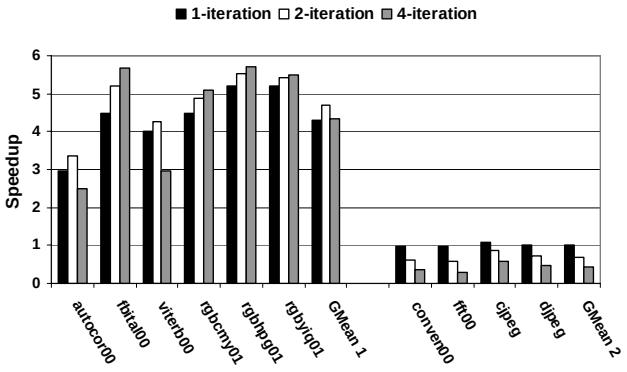

Figure 8: Speedup with thread size of 1, 2 or 4 iteration(s).

It is evident from Figure 8 that the poor performance improvement of the four applications (**conven00, fft00, cjpeg** and **djpeg**) is consistent with the trends shown in the previous sections. Moreover, the exploration of thread-level parallelism in these applications causes performance degradation for a thread size of two or more iterations.

More interestingly, we can see two different trends of performance improvement for the remaining six applications with when thread size is increased. Though the achievable speedup is increasing with increasing thread size for the four applications (**fbital00**, **rgbcmy01, rgbhpg01** and **rgbyiq01**), the other two applications (**autocor00** and **viterb00**) exhibit a different trend. In fact, a thread size of two iterations yields the best performance whereas a thread size of four iterations yields the worst performance in **autocor00** and **viterb00**. Recalling the data in Table 2, we note that the loop trip count of these two applications is sixteen. This implies that an eight-way machine with a thread size of two iterations perfectly balances the load across all the processors and thus amortizes thread-management overhead on a bigger chunk in these two applications. On the other hand, a thread size of 4 or more iterations causes load imbalance and consequently, will have negative impact on the speedup.

From the above results and discussion, we may conclude that the loop trip, the instruction count in each iteration, and the thread-management overhead are the key factors to yield potential performance improvement by exploiting the speculative loop-level parallelism available in the embedded applications considered.

6. Related Work

A lot of research has been devoted to speculative thread-level parallelism. The most common sources of parallelism studied until today are either module-level or loop-level. However, loop-level parallelism offers certain advantages over module-level parallelism. Loops are ideal candidates for extracting parallel threads because of their regular structures and significant coverage on execution time [5, 7, 8, 9, 11, 12]. Moreover, when a loop is encountered, many threads can potentially be spawned at once for many successive iterations [13]. It has been shown in [9] that the loop-iteration spawning policy with an increment predictor and unlimited connectivity architecture is an effective design to exploit speculative thread-level parallelism.

To this date, however, most work has been driven by SPEC benchmark application case studies. Tubella and Gonzalez [11] have proposed the dynamic detection and characterization of loops to obtain speculative threads from a sequential program in the context of multithreaded processors and presented the results using SPEC95 benchmark suite. They have shown that for a feasible configuration with 2, 4, 8 and 16 contexts, their proposed mechanism achieves a TPC (average number of active and correctly speculated threads per cycle) of 1.65, 2.6, 4 and 6.2, respectively [11]. A hardware method for runtime predictability of loop patterns has been evaluated and presented in [2].

The results provided in [2] show that it is possible to correctly predict the number of iterations and the associated path for more than half of the loops visited in SPECint2000 applications. A method of selecting a suitable set of loops for parallelization to improve overall performance has been proposed in [12]. They have shown that a judicious choice of loops can improve the overall performance of SPEC2000 integer benchmarks by as much as 20% [12]. Thus most of the previous works have primarily focused on the applications from SPEC benchmark suite and are particularly suitable for general-purpose and high-end processors for the desktop. On the contrary, our work has evaluated the potential of thread-level parallelism in embedded applications.

Gordon-Ross and Vahid [5] have proposed small on-chip hardware to detect frequent loops. Their work is motivated by the fact that about 85% of the critical regions of code of Powerstone and MediaBench applications are small inner loops or near-inner loops while remaining 15% of the critical regions being subroutines with no inner loops [5]. But our studies have not only detected the performance-critical loops dynamically but also explored the potential of thread-level parallelism out of those loops. Moreover, the objectives of the two studies are quite different from each other.

Only recently, Kejariwal *et al* [7] has presented the challenges in exploitation of loop parallelism in embedded applications. However, our work is significantly different from [7] in several ways. Firstly, they have added instrumentation support through hardware performance counters to analyze the loop coverage which imposes overhead [7] along with a negative impact on the results. But we have calculated the execution-time coverage of the loops without any instrumentation which may have impact on the results. Thus, our work provides a more accurate estimate of the execution-time coverage of the loops. Secondly, it has been shown in [7] that in many programs, innermost loops which are comparatively easier to parallelize may have low coverage. On the contrary, we have shown that eight applications out of ten have a single performance-critical innermost loop with at least 60% coverage of the execution time. Thirdly, they have claimed that hardware and/or software support for data dependence speculation (DDS), control speculation (CS) and data value speculation (DVS) may potentially yield better performance [7]. However, our study has shown that DVS is not needed to get significant speedups. In contrast to their study, we use privatization and parallel reduction and show that it is important to yield significant performance improvements for most of the applications and we provide data for the entire suite of consumer and telecom applications. Finally, we have systematically studied the limits of TLS by analyzing the impact of number of processing elements, thread-management overhead and thread size on the potential performance gain which is not done in [7].

7. Conclusion

In this study, the limits of performance speedup in embedded applications using parallelizing compilers on machines with and without thread-level speculation (TLS) support have been evaluated. We have found that a parallelizing compiler without support for TLS is not adequate to exploit the potential performance improvement of the applications. In fact, only two out of the ten selected applications can be parallelized automatically by the compiler.

We found that TLS support is the key to uncover thread-level parallelism in the applications to achieve substantial speedup. In fact, four applications that the parallelizing compiler was unable to parallelize could be parallelized successfully with TLS support. However, low thread-management overhead along with a coarse-grain thread decomposition strategy is important to get the most out of TLS. Also, from the application perspective, the loop trip and the instruction count per iteration are the two most important characteristics to decide on whether a particular performance-critical loop should be parallelized speculatively. At the end, we have shown that an average speedup of four is achievable on an eight-way multi-core system with TLS support for six out of the ten chosen embedded applications.

Acknowledgments

This research has been partly sponsored by the SARC project under the EU funded FET program. Travel support from the Swedish STINT program is also deeply appreciated.

References

1. V. Agarwal, M.S. Hrishikesh, S. W. Keckler and D. Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA -00)*, pages 248 – 259, 2000.

2. M. Alba and D. Kaeli, ”Runtime Predictability of Loops”, In *Proceedings of IEEE International Workshop on Workload Characterization (WWC-4)*, pages 91 – 98, 2001.

3. D. Burger and T. Austin, “*The SimpleScalar Tool Set Version 2.0*”, University of Wisconsin-Madison, Computer Sciences Department, Technical Report 1342, 1997.

4. EEMBC. http://www.eembc.org/.

5. A. Gordon-Ross and F. Vahid, "Frequent Loop Detection Using Efficient Non-Intrusive On-Chip Hardware", *IEEE Transactions on Computers, Special Issue – Embedded Systems, Microarchitecture and Compilation Techniques*, Vol. 54, Issue 10, pages 1203 – 1215, 2005.

6. Intel® C++ 9.1 Compiler for Linux. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm

7. A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian and H. Saito, "Challenges in Exploitation of Loop Parallelism in Embedded Applications", In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 173 – 180, ACM Press, 2006.

8. D. J. Lilja, "Exploiting the Parallelism Available in Loops", *IEEE Computer*, Vol. 27, Issue 2, pages 13 – 26, 1994.

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, 2000.

10. G. Steffan and T. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization", In *Proc. of Fourth Int. Symp. on High-Performance Computer Architecture*, pp. 2-13, Feb. 1998.

11. J. Tubella and A. Gonzalez, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection", In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA – 4)*, Pages 14 – 23, 1998.

12. S. Wang, X. Dai, K. S. Yellajyosula, A. Zhai and P. Yew, "Loop Selection for Thread-Level Speculation", In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

13. F. Warg and P. Stenström, "Limits on Speculative Module-Level Parallelism in Imperative and Objective-Oriented Programs on CMP Platforms", In *Proc. of Int. Conf. on Parallel Architectures and Compiler Techniques (PACT'2001)*, pages 221-230, Sept. 2001.

14. F. Warg and P. Stenström,, "Improving Speculative Thread-Level Parallelism through Module Run-Length Prediction", In *Proc. of 6th IEEE International Symposium on Parallel and Distributed Processing Symposium*, April 2003.

15. F. Warg, "*Techniques to Reduce Thread-Level Speculation Overhead*", PhD Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2006.

16. Y. Zhang, L. Rauchwerger and J. Torrellas, "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors", In *Proc. of Fourth Int. Symp. on High-Performance Computer Architecture*, pp. 162-173, Feb. 1998.