On the Efficacy of Call Graph-Level Thread-Level Speculation

Arun Kejariwal[§] Milind Girkar[‡] Xinmin Tian[‡] Hideki Saito[‡]
Alexandru Nicolau[†] Alexander V. Veidenbaum[†] Utpal Banerjee[†] Constantine D. Polychronopoulos[¶]
[§]Yahoo! Inc. [‡]Intel Corporation [†]University of California at Irvine [¶]University of Illinois at Urbana-Champaign

ABSTRACT

Thread-level speculation (TLS) has been proposed as a means to parallelize difficult-to-analyze sequential codes. In this paper, we present a realistic measure of the performance potential of call-graph level TLS, using the SPEC [1] CPU2006 benchmark suite and the Intel® Core™2 Duo processor.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems – Measurement techniques
General Terms: Performance, Measurement
Keywords: Thread-level speculation, performance

1. INTRODUCTION

The ever increasing performance requirements coupled with the power constraints has led to the emergence of multi-core systems. Efficient use of such systems is critically dependent on the availability of concurrent software. TLS has been proposed as a means to parallelize difficult-to-analyze (potentially parallel) program regions (see [1] for a detailed listing of prior work in TLS). Although there has been a study on the efficacy of TLS at the innermost loop-level [2], an evaluation of the performance potential of TLS, in modern applications, at call graph-level has not been done so far. To this end, we present an evaluation of the efficacy of TLS at the call graph-level. In particular,

a) Assuming an oracle TLS mechanism, i.e., perfect speculation and zero threading overhead, we evaluate an upper bound on the speedup achievable by applying TLS at the call graph-level.

b) We describe the factors, viz., I/O and recursion (or presence of cycles in the call graph) which limit the applicability of TLS at the call graph-level. We illustrate this with the help of code snippets from SPEC CPU2006 [3].

To the best of our knowledge, this is the *first* limit study of call graph-level TLS using the SPEC CPU2006 benchmark suite. The suite is widely used and considered to be representative of ordinary programs.

2. CALL GRAPH-LEVEL TLS

To extract higher degree of parallelism, call graph-level TLS has been proposed [4] As an example, let us consider the call graph of the function `precompute_arguments` shown in Figure 1 (taken from `403.gcc:calls.c:1482`).

From the figure, we see that `precompute_arguments` calls six functions. Under call graph-level TLS, `precompute_arg--uments` would spawn speculative threads to execute its callees.

[1]Other names and brands may be claimed as the property of others.

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
ACM 978-1-60558-563-5/10/01.

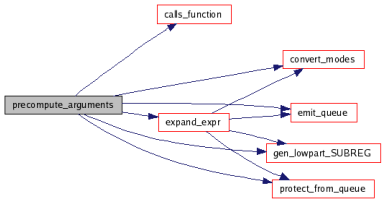

Figure 1: Illustrating TLS at the call graph-level

The function `expand_expr` would then spawn speculative threads to execute its callees, subject to support for nested TLS. In case the callees are embedded in a loop, then different invocations of the same callee are executed by different speculative threads.

2.1 Ideal Case

In this subsection we present an upper bound on the speedup achievable by applying TLS at the call graph level in the ideal case, i.e., assuming zero threading overhead and perfect speculation. We obtained the function coverage – defined as the percentage of the total execution time spent in a given function – profiles, akin to the innermost loop coverage profiles presented in [2]. The benchmarks were compiled using the Intel Fortran/C++ compiler and run on the Intel Core 2 Duo processor (see Table 1 for the system configuration), using the reference data sets.

Processor	Intel®Core™2 Duo Processor (E6600) , 2.4 GHz
Memory	2 GB, DDR 800, dual-channel, non-ECC
L1 D-Cache	32 KB
L1 I-Cache	32 KB
L2 Cache	4 MB
Bus Speed	1066 MHz
Compiler Flags	-O3 -Qansi_alias -Qprof_gen -Qprof_use -QxP -Qipo -Fa
	Microsoft®Windows XP Professional Version 2002 (Service pack 2)

Table 1: Experimental Setup

Analysis of the profiles highlight that the maximum coverage of a single function is small for benchmarks such as `403.gcc` and `445.gobmk`. Thus, in the ideal case, TLS at the call graph-level can potentially yield large speedups – achieved via overlapping the coverage of the different functions in a given call graph (recall that this subject to perfect speculation). For example, in the case of `403.gcc`, it would reduce the execution time to 4% of the total execution time.

However, in practice, the performance gain achievable via call graph-level TLS is limited by a variety of factors such as (but not limited to) presence of I/O and presence of cycles in the call graph level (or recursion). In the rest of this subsection, we discuss these two in detail.

2.2 I/O

One of the factors limiting the applicability of TLS at the call graph-level is the presence of I/O in the candidate function. This stems from the fact that, for example, user input cannot be speculated. Moreover, user input is inherently asynchronous in nature – a user may pause multiple times while providing the input. Likewise, program regions containing read and write file operations are not suitable for TLS. Consequently, functions containing I/O should be excluded from the set of candidate program regions for TLS.

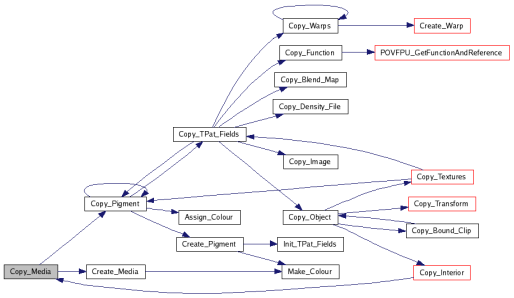

Figure 2: Call graph of `Copy_Media` containing a cycle with multiple nodes

2.3 Recursion

Several hardware mechanisms have been proposed to exploit speculative thread-level parallelism (sTLP). These proposals use a speculative buffer (e.g., see [5, 6]) to store speculative data, track data dependences and correct incorrect execution during roll back. There may exist multiple versions of any given variable. This stems from the fact that concurrently executing speculative threads may produce different versions of the same variable. These versions must be buffered separately. If the speculation is correct, the hardware commits the corresponding data in the speculative buffer to the (non-speculative) memory. The selection of the size of the buffer involves the following trade-off:

a) Typically, the buffers are small memory structures so as to achieve fast access. The need for fast access stems from the following:
   i) Loads and stores account for more than 40% of the total instruction count on an average.
   ii) Every load and store of a speculative thread must be tracked to check for a potential dependence with the non-speculative threads(s).

   However, the small size of the buffer results in buffer overflow whenever the size of speculative state exceeds the buffer capacity. This is common in the case of large speculative threads such as threads executing large functions speculatively.

b) On the other hand, a large speculative buffer helps to uncover higher degree of sTLP by reducing buffer overflow. However, the downside of large buffers is that they have large access times which adversely affects performance. Also, a large buffer has cost implications.

In [7], Garzarán et al. analyzed the trade-offs between the various approaches for buffering memory state in the context of TLS. Recently, Kim et al. proposed a mechanism based on exploiting reference idempotency to reduce buffer overflow [8]. However, they used the train data set which are significantly smaller than the reference data sets. This has a direct implication on the size of the speculative buffer.

The presence of cycles in the call graph relates to speculative buffer size in the following way: unrestricted out-of-order (OOO) spawning of speculative threads may require an infinitely large buffer. Such a scenario might arise when the non-speculative thread incurs a long pipeline bubble and the execution of a speculative thread is "locked" along a cycle in the call graph. For example, let us consider the call graph shown in Figure 2. The function `Copy_Media` is part of the `453.povray` benchmark. From Figure 2 we note that there exist multiple cycles in the call graph – the functions `Copy_Media, Copy_Pigment, Copy_TPat_Fields, Copy_Object` and `Copy_Interior` form a cycle.

Benchmark	Has Recursion	Example
Integer Benchmarks		
400.perlbench	✔	store → store_blessed → store_hook → store
401.bzip2	✗	
403.gcc	✔	simplify_and_const_int → force_to_mode → simplify_shift_const → expand_compound_operation → simplify_and_const_int
429.mcf	✗	
445.gobmk	✔	simple_ladder_attack → simple_ladder_defend → simple_ladder_attack
456.hmmer	✔	reg → regbranch → regpiece → regatom → reg
458.sjeng	✔	develop_node → pn2_eval → develop_node
462.libquantum	✔	quantum_sigma_x → quantum_sigma_x_ft → quantum_sigma_x
464.h264ave	✔	error → flush_dpb → remove_unused_frame_from_dpb → remove_frame_from_dpb → error
471.omnetpp	✗	
473.astar	✗	
483.xalancbmk	✔	ElemTemplate::execute → ElemTemplate::executeChildren → ElemTemplateElement::executeChildren → ElemTemplate::execute
Floating Point Benchmarks		
444.namd	✗	
447.dealII	✔	MappingCartesian::fill_fe_values → MappingCartesian::compute_fill → MappingCartesian::fill_fe_values
450.soplex	✔	soplex::SPxDefaultRT::selectEnter → soplex::SPxDefaultRT::selectEnter → soplex::SPxDefaultRT::selectEnter
453.povray	✔	Copy_Textures → Copy_Thormal → Copy_TPat_Fields → Copy_Object → Copy_Textures
482.sphinx3	✔	mdef_free_recursive_lc → mdef_free_recursive_lc

Table 2: Benchmarks in SPEC CINT2006 and (C/C++ based) CFP2006 which have recursive code

For a practical assessment of the performance potential of TLS at the call graph-level, functions which are part of a cycle in the call graph should be excluded from the set of candidate functions for TLS. In addition, all the callers of such functions should also be excluded from the set! Limiting the speculation depth can potentially be of help in this regard. However, determining the depth at run time is non-trivial.

Table 2 lists all the benchmarks CINT2006 and C/C++ CFP2006 benchmarks which have at least one cycle in their call graphs. An example cycle in such benchmarks is also given. From the table we see that majority of the benchmarks contain recursive code.

2.4 Upper Bounds

Table 3 reports the coverage of the top 5 hot functions which are not part of a cycle in the call graph and none of their callees *and* the functions in their respective call graphs are part of a cycle (in other words, there should not exist a path between the given function and a cycle in the call graph). For example, in Figure 2, the function `Create_Pigment` is not a part of a cycle and neither are its callees. In addition, the candidate function or its callees should not contain library calls. Further, based on the discussion presented in subsection 2.2, we excluded the functions which contain I/O. The coverage reported in the table serves as an upper bound on the applicability of TLS at the call graph level, assuming no conflict between the different speculative threads and a zero threading overhead. Obtaining tighter bounds based on static analysis and/or run time profiling of dependences between different functions and accounting for the impact of threading overhead on the efficacy of call graph-level TLS is a subject of future work.

Integer Benchmarks			
Benchmark	Coverage	Benchmark	Coverage
400.perlbench	11.63	403.gcc	8.08
445.gobmk	6.88	456.hmmer	≈ 1
458.sjeng	10.65	462.libquantum	8.68
464.h264ref	50.41	483.xalancbmk	5.21
Floating Point Benchmarks			
Benchmark	Coverage	Benchmark	Coverage
447.dealII	17.48	450.soplex	9.96
453.povray	17.75	482.sphinx	14.03

Table 3: Performance potential of call graph level-TLS

3. REFERENCES

[1] A. Kejariwal and A. Nicolau. Reading list of performance analysis, speculative execution. http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf.

[2] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *PPoPP*, 2007.

[3] SPEC CPU2006. http://www.spec.org/cpu2006.

[4] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT*, 1998.

[5] M. Franklin. Multi-version caches for multiscalar processors. In *Proceedings of 1st International Conference on High Performance Computing*, 1995.

[6] S. Gopal, T. N. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *HPCA*, 1998.

[7] M. J. Garzarán, M. Prvulovic, M. Llabería J, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM TACO*, 2(3):247–279, 2005.

[8] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM TOPLAS*, 28(5):942–965, 2006.