Breaking Cyclic-Multithreading Parallelization with XML Parsing

Simone Campanoni Svilen Kanev Kevin Brownell Gu-Yeon Wei David Brooks
Harvard University
{xan,skanev,brownell,guyeon,dbrooks}@eecs.harvard.edu

1. Introduction

HELIX-RC, a modern re-evaluation of the cyclic-multithreading (CMT) compiler technique [6], extracts threads from sequential code automatically. As a CMT approach, HELIX-RC gains performance by running iterations of the same loop on different cores in a multicore. It successfully boosts performance for SPEC CINT benchmarks previously considered unparallelizable (Table 1), assuming 16 Intel Atom cores with a special-purpose interconnect [3]. However, this paper shows there are workloads with different characteristics, which even idealized CMT cannot parallelize.

We identify how to overcome an inherent limitation of CMT for these workloads. CMT techniques only run iterations of a single loop in parallel at any given time. We propose exploiting parallelism not only within a single loop, but also among multiple loops. We call this execution model *Multiple CMT (MCMT)*, and show that it is crucial for auto-parallelizing a broader class of workloads.

To highlight the need for MCMT, we target a workload that is naturally hard for CMT – parsing XML-structured data. XML parsing has been characterized as highly input-dependent, with a complex callgraph structure [4]. The particular implementation that we used (`libxml2` [11]) adds further characteristics that complicate blindly applying a CMT approach. First, it spends a significant amount of execution in recursive calls or in non-natural loops, limiting the application scope of CMT (which exploits natural loop iterations). Second, it spends a rather small fraction of execution in small loops (which are preferred by HELIX-RC) – `libxml2`: 15%, SPEC CINT on average: 90%.

The rest of the paper starts by describing the importance of XML parsing and the code characteristics of `libxml2`. Then, after showing that HELIX-RC comes short for this library, we demonstrate that even idealized CMT-like techniques are not effective. On the other hand, an MCMT prototype speeds up the parsing task by up to 3.9× on 4 cores.

Table 1: Code complexity (approximated by # lines of code) and achieved speedup between SPEC2000 and libxml2.

Benchmark		LOC	HELIX-RC Speedup [3]	Parallel loop coverage (%)
CFP2000	177.mesa	42,491	15.1×	99%
	179.art	1,036	10.5×	99%
	183.equake	1,042	10.1×	99%
	188.ammp	9,805	12.5×	99%
CINT2000	164.gzip	5,630	3.0×	98.2%
	175.vpr	11,300	6.1×	99%
	181.mcf	1,482	8.7×	99%
	197.parser	7,763	7.3×	98.7%
	256.bzip2	3,235	12.0×	99%
	300.twolf	17,875	7.6×	99%
libxml2		**170,893**	**1.02×**	**48%**

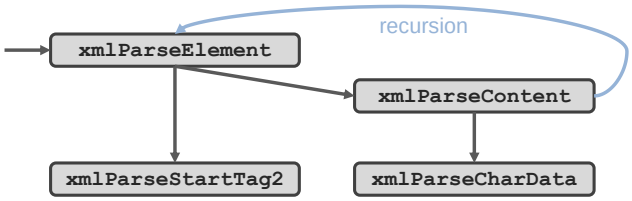

Figure 1: Simplified libxml2 parsing callgraph.

2. XML parsing

XML has established itself as the *de facto* standard for exchanging semi-structured data in an interoperable and standardized way, leading to several efforts to speed up parsing by manual parallelization [7, 12, 13], or even by hardware acceleration [8]. Mobile browsers also spend considerable resources on the very similar task of HTML parsing and DOM tree construction. As an example, on the Exynos 5410 SoC, Chromium spends 17% of its execution time and 16% of its energy on this task [15].

In order to understand why CMT performs poorly on XML parsing, the rest of this section takes a closer look at the parser implementation in `libxml2` (Figure 1). We find code properties that limit the amount of parallelism between iterations of the same loop – recursion, non-natural loops and loops with a single iteration.

Code characteristics

The parser has a very commonly executed recursive cycle (between the functions `xmlParseElement` and `xmlParseContent`) for inspecting XML subelements. It builds up significant execution time without any loops whatsoever, limiting the amount of parallelism that loop-centric CMT can extract. Furthermore, not all hot loops are natural, and easily identifiable by a compiler. Such non-natural loops are the result of heavy manual optimization with `goto` statements (mostly in the function `xmlParseStartTag2`), some of which jump in the middle of a loop body. These jumps violate the property that a compiler uses to to identify a loop: the header of the loop must pre-dominate the loop body. Similarly to recursion, this renders CMT ineffective.

The balance between the functions described above (and hence the quality of CMT parallelization) is strongly dependent on the shape of the input XML tree[1]. The more deeply nested the tree is, the more time is spent on the recursion between `xmlParseElement` and `xmlParseContent`, and therefore, less time is spent in CMT-targetable code. Similarly, the more elements a tree has, the more time is spent in `xmlParseStartTag2`, including in its non-natural loops. On the contrary, XML with more leaf data causes more invocations of `xmlParseCharData`, whose callees contain tens of natural loops, leading to potentially better parallelization.

[1] Similar dependences have been observed for manual parallelization, leading to a pre-parsing step to discover the XML structure [7, 13].

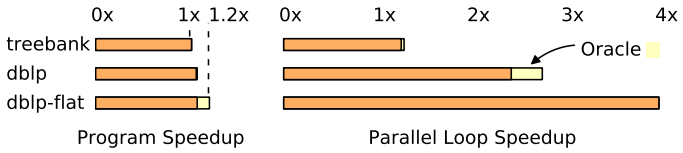

Figure 2: Even oracle analyses cannot compensate for the lack of loop iteration parallelism exploited by CMT.

Based on these insights, we use three inputs with different shapes of the XML tree. The first two, `treebank` and `dblp` [9], have trees with average nesting levels of 7.9 and 2.9. The last one, `dblp-flat`, is explicitly flattened, with all elements being direct children of the root. We expect to obtain best performance for `dblp-flat` and worst for `treebank`.

Profiling data indicates that the code characteristics described above significantly limit the program coverage of natural loops. Even in the best-case input (`dblp-flat`, no recursion), 28% of execution time is spent in the non-natural loops of `xmlParseStartTag2`, which are not considered by CMT (in addition to 7% in initialization and de-initialization code). As expected, recursion for deeply nested XML causes lower coverage – time in natural loops goes down from 65% for `dblp-flat` through 60% for `dblp` to only 48% for `treebank`. Surprisingly, most loops that cover this 48-65% only have a single iteration each. For example, some iterate over namespaces and attributes, which are not abundant in our inputs.

3. Limitations of CMT

HELIX-RC does not extract enough parallelism from `libxml2`. We ran HELIX-RC (as fully described previously [3]) on the `libxml2` parser, assuming a four-core Atom-like platform. We model speedup at the compiler intermediate representation (IR) level, modelling the cost of IR instructions. Although the speedup within parallel loops gets up to 3.5×, they only cover between 9% and 12% of execution, resulting in low overall program speedups (1.02×–1.08×) shown in Figure 2. The low coverage is due to recursion, non-natural loops, and single-iteration loops. Therefore, other CMT techniques, like HELIX [1, 2], STAMPede [10], Stanford Hydra [5], DOACROSS, and DOALL [6, 14], cannot succeed in parallelizing `libxml2` as well.

Even idealized CMT results in low performance. To estimate the potential of ideal CMT, we measure performance gained by HELIX-RC after replacing conservative code analyses with oracle information. The code analyses that get replaced by oracles are: data dependence, control dependence, induction variable, and function pointer. Therefore, the only code in loops that runs sequentially consists of unpredictable read-after-write dependence chains. The oracle bars in Figure 2 show virtually no improvement over using realistic analyses simply because of the lack of loop iteration parallelism.

4. Beyond CMT

To overcome the inherent limitation of CMT, Multiple CMT (MCMT) distributes iterations of *multiple* loops among cores. MCMT runs multiple loops concurrently on different cores by spreading each loop's iterations on a different core subset. A prototype runtime that implements MCMT achieves speedup between 3.0 and 3.9× on `libxml2` parsing, assuming a four-core platform (Figure 3, third column).

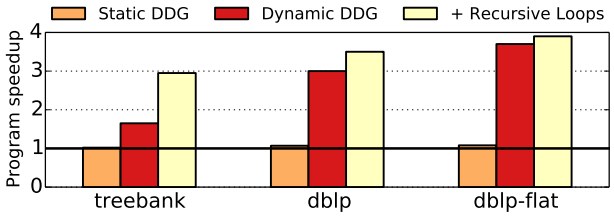

Figure 3: Exploiting parallelism among multiple loops is essential to gain performance.

In more detail, MCMT includes a runtime which starts executing non-loop code serially. Once it encounters a natural loop, it dispatches its iterations among a subset of cores. Without waiting for these iterations to complete, the runtime continues executing subsequent code. If this code contains another loop, its iterations are also dispatched on different cores, concurrently with those of the first loop. For correctness, data and control dependences are satisfied through normal synchronization. Finally, we optimistically assume zero overhead for the dispatch decisions.

While MCMT has potential, it requires run time support for accurate data dependence information. This is confirmed by the large difference between the first two sets of bars in Figure 3 – using static and dynamic data dependence graph (DDG). This is expected because obtaining high accuracy on data dependence analysis is a challenge on a large codebase. Moreover, because MCMT explicitly targets multiple loops, its DDG analysis cannot look into single loops in isolation, and must potentially address the entirety of a large codebase. This makes static analyses impractical for codes the size of `libxml2` (Table 1). Hence the need for a run time approach potentially based on speculative multithreading [10].

One last optimization for MCMT allows it to capture recursion better. In this more general model, iterations of different invocations of the same loop can also run in parallel. The effect of this optimization is stronger on more recursive inputs – Figure 3, the gap between the second and the third column.

References

[1] S. Campanoni *et al.*, "Helix: Automatic parallelization of irregular programs for chip multiprocessing," in *CGO*, 2012.
[2] S. Campanoni *et al.*, "HELIX: Making the extraction of thread-level parallelism mainstream," *IEEE Micro*, 2012.
[3] S. Campanoni *et al.*, "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs," in *ISCA*, 2014.
[4] T. Cheung *et al.*, "XML Document Parsing: Operational and Performance Characteristics." *IEEE Computer*, 2008.
[5] L. Hammond *et al.*, "The Stanford Hydra CMP," in *IEEE Micro*, 2000.
[6] A. R. Hurson *et al.*, "Parallelization of DOALL and DOACROSS Loops - A Survey," *Advances in Computers*, 1997.
[7] W. Lu *et al.*, "A parallel approach to xml parsing," *ICGC*, 2006.
[8] J. V. Lunteren *et al.*, "XML accelerator engine," in *International Workshop on High Performance XML Processing*, 2004.
[9] G. Miklau, "UW XML Repository," 2006.
[10] J. G. Steffan *et al.*, "The STAMPede approach to thread-level speculation," *ACM Trans. Comput. Syst.*, 2005.
[11] D. Veillard, http://xmlsoft.org.
[12] C.-H. You and S.-D. Wang, "A Data Parallel Approach to XML Parsing and Query," *High Perf. Computing and Communications*, 2011.
[13] Y. Zhang *et al.*, "Speculative p-DFAs for parallel XML parsing," in *International Conference on High Performance Computing*, 2009.
[14] H. Zhong *et al.*, "Uncovering hidden loop level parallelism in sequential applications," in *HPCA*, 2008.
[15] Y. Zhu *et al.*, "WebCore: Architectural Support for Mobile Web Browsing," in *ISCA*, 2014.