

Complementing User-Level Coarse-Grain Parallelism with Implicit Speculative Parallelism^{*}

Nikolas Ioannou, and Marcelo Cintra[†]
School of Informatics
University of Edinburgh
{nikolas.ioannou@|mc@staffmail}.ed.ac.uk

ABSTRACT

Multi-core and many-core systems are the norm in contemporary processor technology and are expected to remain so for the foreseeable future. Programs using parallel programming primitives like *PThreads* or *OpenMP* often exploit coarse-grain parallelism, because it offers a good trade-off between programming effort versus performance gain. Some parallel applications show limited or no scaling beyond a number of cores. Given the abundant number of cores expected in future many-cores, several cores would remain idle in such cases while execution performance stagnates. This paper proposes using cores that do not contribute to performance improvement for running *implicit* fine-grain *speculative* threads. In particular, we present a many-core architecture and protocol that allow applications with coarse-grain explicit parallelism to further exploit implicit speculative parallelism within each thread. Implicit speculative parallelism frees the programmer from the additional effort to explicitly partition the work into finer and properly synchronized tasks. Our results show that, for a many-core comprising of 128 cores supporting implicit speculative parallelism in clusters of 2 or 4 cores, performance improves on top of the highest scalability point by 41% on average for the 4-core cluster and by 27% on average for the 2-core cluster. These performance improvements come with an energy consumption that is close to – and sometimes better than – the baseline. This approach often leads to better performance and energy efficiency compared to existing alternatives such as Core Fusion and Frequency Boosting. We also investigate the tradeoffs between explicit and implicit threads as input dataset sizes vary. Finally, we present a dynamic mechanism to choose the number of explicit and implicit threads, which performs within 6% of the static oracle selection of threads.

Categories and Subject Descriptors

C.4 [Performance of Systems]: Design studies; C.1.4 [Processor Architectures]: Parallel architectures

^{*}This work was supported in part by EPSRC under grant EP/G000697/1 and the EC under grant HiPEAC IST-004408.

[†]M. Cintra is currently on sabbatical leave at Intel Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MICRO'11 December 3-7, 2011, Porto Alegre, Brazil
Copyright 2011 ACM 978-1-4503-1053-6/11/12 ...\$10.00.

General Terms

Design, Experimentation, Performance

Keywords

Many-core architecture, thread-level speculation, thread-level parallelism

1. INTRODUCTION

With the shift toward multi- and many-cores, programmers can no longer enjoy steep performance improvements for free with every new generation of processors. Instead, parallel programming has to be employed both for programs written from scratch and for legacy code in order to exploit this new hardware. However, parallel programming is often hard and error prone, specially when addressing fine-grain threading which involves complex synchronization, communication, data partitioning, and scheduling [25]. Thus, programmers often stay away from fine-grain parallelism and concentrate their efforts in exploiting parallelism at a coarser granularity. Coarse-grain parallelism offers a good compromise between development effort and performance, and is often the first step exploited by programmers as they incrementally parallelize and performance tune their programs. Typical examples of such types of parallelism can be implemented via *PThreads* and *OpenMP*.

Given this focus on coarse-grain parallelism, applications often show limited or no scaling when executed in a large number of cores. Typical reasons for this are, among others, large critical sections leading to serialization in the presence of many threads, load imbalance between threads, and communication and coherence overheads [13, 15]. On the other hand, this focus on coarse-grain parallelism means that there is often room for opportunistically exploiting further degrees of fine-grain parallelism [13, 15].

In this paper we propose allocating cores beyond the application's scalability limit to exploit implicit speculative parallelism within individual explicit threads. By running implicit speculative threads through *thread-level speculation (TLS)* [14, 22, 29, 39] performance can be improved beyond the application's scalability limit for a given input dataset. Moreover, given the guaranteed sequential semantics of the TLS protocol, this further parallelization is transparent to the programmers so that they do not have to struggle to further partition and debug the parallel code. In fact, with TLS it is possible to exploit whatever degree of parallelism exists within the coarse-grain explicit threads even in the presence of data dependences.

Under our scheme the implicit speculative threads operate *within* explicit parallel threads with support for both types *simultaneously* in a nested fashion. Prior work that proposed architectures with support for both TLS and explicit threads [33, 34] could only accommodate either type

```

1 /*IS Kernel*/
2 Do for i=1 to Imax {
3   if(master_thread)
4     modify_the_sequence_of_keys;
5   BARRIER(all_procs);

7   for( i=0; i<NUMKEYS; i++ )
8     compute_rank_of_each_key_locally;

10  lock(CS_lock);
11  update_global_key_array;
12  unlock(CS_lock);
13  BARRIER(all_procs);

15  if(master_thread)
16    perform_partial_verification;
17 }

```

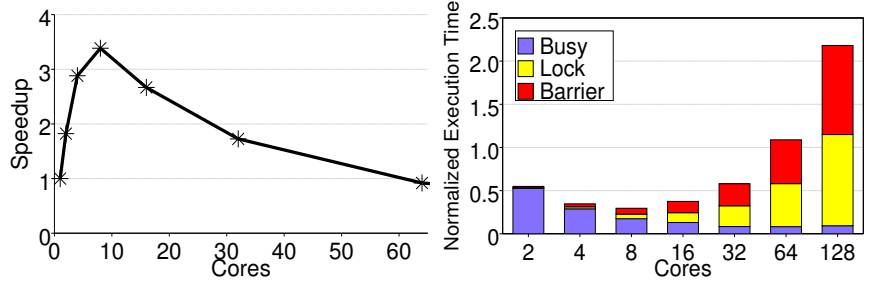


Figure 1: Scaling behavior of the IS benchmark from the NAS Parallel Benchmarks [1]: (a) Kernel source code. (b) Speedup scaling. (c) Breakdown of execution time.

at a time by switching between modes. We have developed a combined and nested coherence plus TLS protocol that provides coherence across explicit parallel threads and simultaneously provides TLS across multiple groups of implicit speculative threads, where each group is associated with a single explicit thread. This protocol is similar in spirit to previously proposed ones [31, 44] that allow nesting of transactional memory and coherence, but it requires some specific mechanisms in order to accommodate the differences in behavior between TLS and transactional memory.

Figure 1 shows a simple example case study. Increasing the number of cores beyond eight causes performance to decrease significantly (Figure 1b). This is due to a large critical section whose relative execution time increases with the number of cores, as can be seen in Figure 1c. In the critical section (lines 10–12 in Figure 1a) each thread simply adds its local keys to the global key array in a *for* loop. This loop is amenable to parallelization and doing so could reduce the time spent in the critical section. Unfortunately, explicitly parallelizing this critical section requires writing some non-trivial code that allows threads to be dynamically detected when waiting at the critical section and then dynamically join the thread that is inside the critical section to assist it in performing its work.

Our approach has similar aims as Core Fusion [20] and Frequency Boosting [18] in that they all attempt to pre-allocate or shift resources to a subset of cores in order to accelerate sections of a parallel code that do not scale well. However, they all differ in the source of the acceleration and hardware support. In our evaluation we quantitatively compare these approaches in terms of performance and energy efficiency.

The main contributions of this paper are:

- We are the first to evaluate implicit speculative parallelism *on top* of explicit parallelism as a means to improve performance in traditional multi-threaded applications that exhibit poor scaling.
- We discuss the architectural requirements for a system supporting implicit and explicit threads concurrently and evaluate such a many-core architecture.
- We present detailed analysis of performance bottlenecks in a set of multithreaded applications and evaluate their behavior in the presence of different input datasets.
- We present a hill-climbing approach that dynamically selects the number of explicit and implicit threads for a class of parallel programming style.

Our experimental results show that complementing parallel programs with implicit speculative mechanisms offers significant performance improvements for a large and diverse

set of parallel benchmarks. For a many-core comprising of 128 cores, performance improves on top of the highest scalability point by as much as 98%, and 41% on average, for a 4-way cluster and by as much as 42%, and 27% on average, for a 2-way cluster¹. These performance improvements come with virtually no increase in total energy consumption. Compared to the alternative – Core Fusion and Frequency Boosting – our approach often leads to higher performance with consistently lower energy consumption. Furthermore, our mechanism to choose the number of explicit and implicit threads performs within 6% of the static oracle thread selection.

The rest of the paper is organized as follows: Section 2 details our approach and its architectural support; Section 3 presents the evaluation methodology; Section 4 discusses the experimental results; Section 5 presents the related work; and Section 6 concludes the paper.

2. EXPLOITING IMPLICIT SPECULATIVE PARALLELISM IN EXPLICITLY PARALLEL APPLICATIONS

2.1 General Idea

The key realization that we exploit in this paper is that explicitly parallel applications with user-level coarse-grain threads are often limited in scalability sooner or later. Further decomposing the threads into smaller tasks beyond that point is often very difficult or futile. As future many-cores are expected to have tens to a few hundred cores, many applications will have an execution scalability cap below the total number of cores available on chip. The key idea of this paper is to complement the explicit coarse-grain threads with fine-grain implicit speculative threads. In this scenario the programmer has to parallelize, debug, and performance tune the application only up to a desired cost-benefit point with coarse-grain threads. Further fine-grain parallelism is then exploited *from within each coarse-grain thread* by the hardware using thread-level speculation (TLS). As TLS provides sequential semantics, parallelism is exploited implicitly and transparently without the programmer having to worry about work partitioning, communication, synchronization, and scheduling. We note that since speculation is applied at the fine granularity of loops and procedure calls within a coarse-grain explicit thread, its effect is quite different from that of simply dividing the amount of work to be done by the explicit thread.

¹We use the term *n*-way cluster (or *n*-way TLS) to refer to a scheme that partitions a given explicit thread into *n* implicit threads.

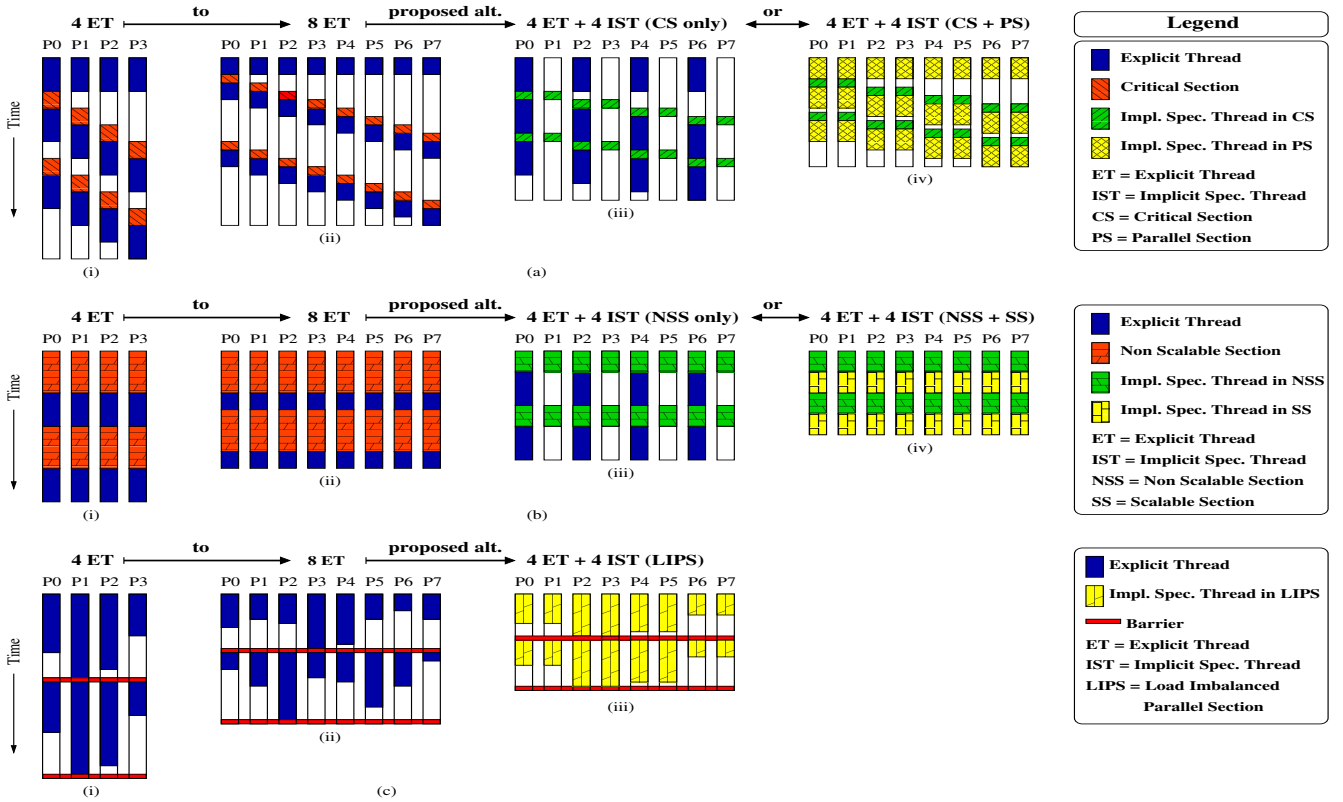


Figure 2: Sources of bottlenecks and possible uses of implicit threads: (a) Application with a large critical section. (b) Application with regions that do not scale proportionally with the dataset. (c) Applications with load imbalance.

Figure 2 illustrates this idea. Figure 2a shows the case of an application dominated by critical sections. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 2a-i to Figure 2a-ii) successfully reduces the execution time of the parallel and critical sections, but overall scalability is limited by the serialization of the critical sections². We propose to use the underutilized cores to run the critical section in TLS mode. By allocating resources for speculative threads (Section 2.2) we effectively trade off reduced execution time of the parallel sections for reduced serialization of the critical sections and possibly reduction of the execution time of the critical sections (Figure 2a-iii). Figure 2b shows the case of an application where the parallel section is divided into one region with execution time proportional to the dataset and one with fixed execution time. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 2b-i to Figure 2b-ii) successfully reduces the execution time of the dataset proportional region only. Again, by using the additional cores to run these sections in TLS mode we can achieve some parallelization of the non-scalable regions (Figure 2b-iii). Another case we found in our experiments is that shown in Figure 2c of an application where static coarse-grain work partition leads to increased load imbalance. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 2c-i to Figure 2c-ii) successfully divides the work done but not in equal portions. By running the original explicit threads in fine-grain TLS mode we can achieve a more even partition

²In some cases, as that of IS of Figure 1, the problem is worsened by the fact that the critical section time does not scale with the further work partition and the serialization problem is compounded with the fixed time of the critical section

of the work, leading to less load imbalance (Figure 2c-iii). Finally, we note that we can further improve on this simple model by exploiting implicit speculative threads also in the parallel sections of Figure 2a and in the dataset proportional regions of Figure 2b, as shown in Figures 2a-iv and 2b-iv respectively.

Figure 3 illustrates the performance behavior we hope to achieve with our proposed approach. In this example a baseline system with only explicit parallel threads scales in region **A** but stops scaling beyond 16 cores. In this region the speedups of both a 2-way and a 4-way TLS schemes are likely below that of the baseline, since TLS, with its overheads and limited coverage, cannot compete with the performance gains from more explicit threads. However, after the baseline stops scaling, a 2-way TLS can potentially still provide performance gains for yet another doubling of the number of cores, as shown in region **B**. After this point we expect the speedup curve of the 2-way TLS system to behave similarly to that of the baseline, since a 2-way TLS can only provide at best a fixed performance boost over a corresponding baseline system with the same number of cores. On the other hand, a 4-way TLS performance curve will take longer to catch up with the baseline and a 2-way TLS system but, ideally, this system can still provide performance gains after the 2-way TLS stops scaling for yet another doubling of the number of cores, as shown in region **C**. Again, we expect that after this point (not shown in the figure) the speedup curve of the 4-way TLS system will also behave similarly to that of both the baseline and a 2-way TLS. Previous work on TLS has shown that it does not scale very well beyond 4 or 8 cores, which means that our proposed approach will only “buy” performance boosts up to systems with 4 to 8 times the number of cores. However, the goal of our approach is

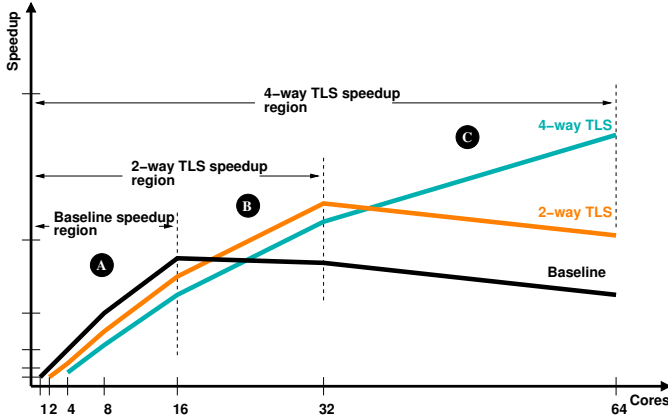


Figure 3: Expected speedup behavior with and without implicit speculative threads.

not to provide indefinite scalability, but to allow applications with poor scalability to better exploit the few hundreds of cores expected to become available in many-core systems by the end of the CMOS road map.

Supporting this idea requires some small changes to the hardware. Unlike previous TLS schemes that attempted to achieve scalable performance solely through TLS, in our scheme it is sufficient that TLS be supported only in groups of small numbers of cores. In fact, if one expects most explicitly parallel applications to scale at least up to half or a quarter of the number of cores in the system, then all one needs is groups of 2- or 4-way TLS, respectively. In addition to supporting TLS within groups of cores, our scheme also requires some small changes to the TLS and coherence protocols in order to allow them to operate *simultaneously* in a nested fashion: coherence at the outer layer across groups of cores running explicit threads, and TLS at the inner layer within groups of cores running implicit speculative threads.

2.2 Many-Core Architecture

As explained in Section 2.1, our proposal to deal with the problem of limited scalability of explicitly parallel applications is to speculatively parallelize portions of the explicit threads. For this it suffices to support TLS within small groups of cores, or *TLS domains*. A natural physical organization is then to partition the many-core in several *tiles* as shown in Figure 4a, where each tile is an independent TLS domain, as shown in Figure 4b. Given the small number of cores per tile, cache coherence can be easily enforced within tiles by a snooping protocol on a bus, although directory based approaches are also possible. Cache coherence across tiles is enforced by a distributed directory protocol, which also interfaces with the intra-tile coherence protocol layer to build a fully coherent hierarchical system such as the one in [26]. This clustered organization is in line with expected trends for many-core systems [23] and can be already partially seen in the recent SCC system from Intel [16], which has multi-core tiles for routing purposes.

Each physical cluster corresponds to a TLS domain and TLS can be easily enforced with a snooping TLS protocol, such as [6], although directory-like approaches, such as [22], are also possible if coherence is also enforced by directories within clusters. Again, there is no need to fully support TLS across clusters, although some interaction between the domains is necessary, as explained in more detail in Section 2.3.

2.3 Nested Coherence and TLS Protocol

We base our architecture on a tiled many-core where each tile comprises a cluster of two or four cores, and where TLS is enforced within each cluster and coherence is enforced system wide. Simultaneously supporting coherence and multiple, independent TLS domains in a nested way imposes some restrictions on the flavor of TLS protocol used and also requires some mild changes to both protocols. The overall organization of the protocols is shown in Figure 4c, with a *TLS protocol layer* operating in each TLS domain underneath a *Coherence protocol layer* that operates across domains when running both explicit and implicit threads and additionally operates within domains when running only explicit threads.

2.3.1 TLS Protocol

The TLS protocol we use performs *eager conflict detection* at a mixed *word and line granularity* with *forwarding* and *lazy version management and commit*. An eager conflict detection policy (e.g., [38]) means that speculative threads are squashed as soon as a data dependence violation is detected. Conflicts occur when a speculative thread reads a value that is later modified. These can occur between a speculative thread and its predecessors in the same TLS domain (case 4 in Figure 4c) and also between a speculative thread and the non-speculative thread in other TLS domains (case 5+6 in Figure 4c). The later case has to be treated as a violation (leading to the squash of the speculative thread) to guarantee that a later speculative thread (in sequential order) does not consume an earlier value than an earlier speculative thread. Figure 5a depicts this problem: not squashing speculative thread $T_{i,2}$ at the time of the invalidation (action 4 in the figure) leads to incorrect execution semantics. Thus, in the nested protocol incoming invalidation requests from other clusters lead to squashes of speculative threads (note that non-speculative threads, such as $T_{i,0}$ in the figure, do not have to be squashed). A similar approach to nesting is followed by transactional memory systems that enforce strong isolation between transactional and coherent threads [4].

We have chosen to implement conflict detection at the granularity of words within each TLS domain as this has been shown to provide better performance due to the absence of false violations leading to squashes [9, 34]. However, for the conflict detection between speculative threads in one TLS domain and non-speculative threads in another TLS domain, as described above, we must perform conflict detection at the granularity of whole cache lines. This is because the coherence protocol operates at the granularity of lines.

Forwarding [9] is supported entirely within a TLS domain when a speculative thread loads a value that has been previously modified by a predecessor speculative thread (case 2 in Figure 4c). Speculative loads that do not find a version within the TLS domain must cross to the coherence layer (Section 2.3.2) in order to both obtain the return value and allow the identification of potential conflicts with coherent threads, as explained above (case 1 in Figure 4c).

A lazy version management and commit policy (e.g., [6, 38]) means that writes by speculative threads are not merged with the non-speculative state until the thread becomes non-speculative and commits. In the nested protocol committing involves merging the state with the non-speculative state within a TLS domain and also with the coherent state across domains. We note that the alternative, eager version management with logs [32] leads to subtle interactions between speculative and non-speculative (i.e., coherent) threads as

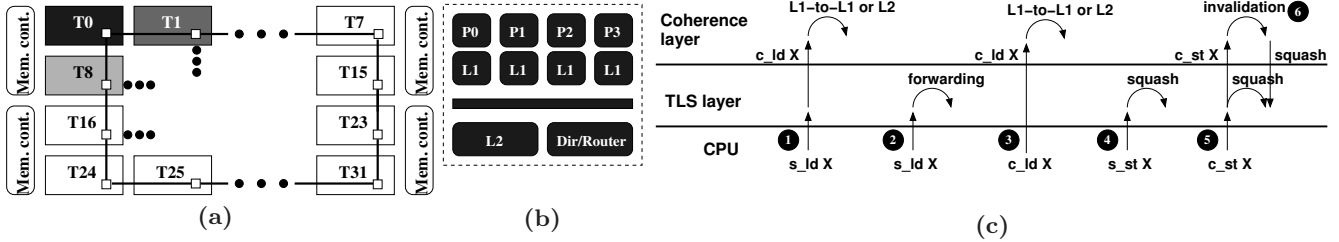


Figure 4: Organization of tiled many-core architecture: (a) Many-core organization. (b) Tile organization. (c) Hierarchical protocol view (s_ld: speculative load, s_st: speculative store, c_ld: coherence load, c_st: coherence store).

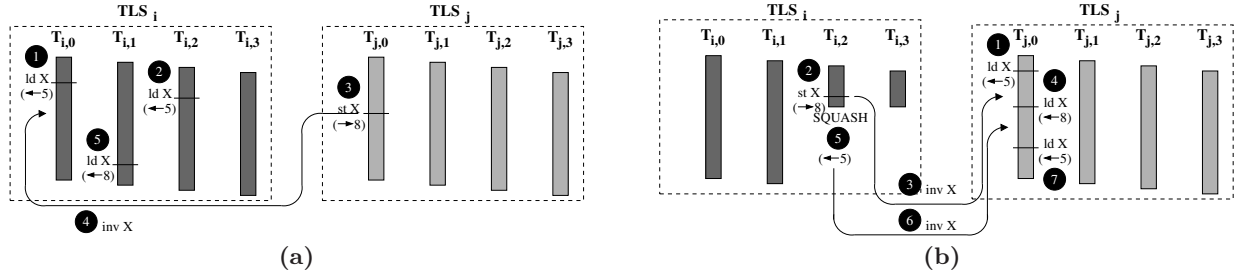


Figure 5: Example of interaction between coherence and TLS protocols: (a) Incorrect handling of coherence invalidation. (b) Incorrect log-based rollback.

previously identified in [4]. The problem is depicted in Figure 5b: after a squash of thread $T_{i,2}$ undoing the write to X leads to an inconsistent state in the non-speculative thread $T_{j,0}$ since coherence transactions (invalidation 3 following store 2 in Figure 5b and subsequent load 4 of the new value) cannot be undone. The lazy commit alternative avoids this by not allowing speculative stores to be communicated across clusters before thread commit. Finally, commits must appear to be atomic and, as mentioned above, involve not only the TLS domain but the entire system. Thus, lazy commit is reasonably straightforward in bus based systems (e.g., [6, 38]), but is more involved in directory based systems [7]. We follow an approach similar to the one in [7] where committing threads must obtain a special “commit token” from all directories associated with their read and write sets. After the token has been acquired the thread can safely commit all the cache lines in its write set, with the directories affected sending invalidation messages to any sharers. A speculative thread with the lower id always succeeds in committing when there is a write conflict with another speculative thread, thus avoiding deadlock.

2.3.2 Coherence Protocol

The coherence protocol used is *MESI* with *coarse-grain* directory state *per cluster* and a “pseudo-CPU” approach to interface *snooping coherence within each cluster and directory coherence across clusters* [26]. A coarse-grain directory with “pseudo-CPU” approach means that directories do not store coherence information for every core in the clusters, but only a summary information per cluster. This is convenient when running a coherent plus multiple speculative threads in each cluster since the TLS behavior does not have to be exposed to remote directories and the global coherence protocol. Instead, the “pseudo-CPU” (whose part is played here by the directory controller in each cluster - Figure 4b) is responsible for propagating coherence requests when running only explicit threads or “translating” back and forth between coherence and TLS transactions. The local directories are also responsible for handling the speculative commit scheme of [7], as described in Section 2.3.1.

2.4 Dynamically Choosing the Number of Explicit and Implicit Threads

Finding the exact scalability tipping point for a particular application and input dataset of interest is commonly done in the HPC community by trials with increasing number of cores [12]. Ideally, however, one would like a mechanism for automatically finding this tipping point on-the-fly and then choosing whether to employ implicit threads on top of explicit ones. For this purpose, we have developed a simple hill climbing algorithm (Algorithm 1) and implemented a prototype in the Omni OpenMP system [24]. This dynamic approach works for applications that are amenable to Dynamic Concurrency Throttling (DCT) [10]. The algorithm begins by choosing the initial number of threads to evaluate using a heuristic: If the maximum number of iterations in all the *omp for* loops for the current parallel region can be determined statically and is less than or equal to the number of available cores (128) we set the initial number of threads to that value; otherwise we set it to 32, which is a value that we empirically found to require the least amount of training time for the evaluated workloads (lines 2-5 in Algorithm 1). We then evaluate this initial thread count and perform a hill climbing search until it detects a slowdown in the execution time of the same parallel region or it encounters a thread count lower than 1 or greater than 128 (lines 7-17 in Algorithm 1). Note that the hill climbing searches toward lower thread counts first as a repercussion of setting the initial thread count equal to the maximum iteration count. After settling on the number of explicit threads, a simple empirical heuristic chooses whether to employ TLS or not (lines 18-19 in Algorithm 1). The heuristic is based on our observation that the tipping point on the number of explicit threads remains (mostly) unchanged when enabling TLS.

3. EVALUATION METHODOLOGY

3.1 Simulation and Compilation Environment

We conduct our experiments using the SESC simulator [37]. The access latencies and power consumption of the core and memory hierarchy are obtained using CACTI [41] and

| Core | |
|-------------------------|---------------------------------------|
| Frequency | 3GHz |
| Fetch/Issue/Retire | 4/4/4 |
| L1 ICache (IL1) | 32KB, 2-way, 2-cycles, 64B |
| L1 DCache (DL1) | 32KB, 4-way, 3-cycles, 64B |
| IL1/DL1 MSHR Entries | 10/16 |
| I-Window/ROB | 80/96 |
| Branch Predictor | 16Kbit Hybrid |
| BTB/RAS | 1K entries, 2-way / 32 entries |
| Tile/System | |
| Number of Cores | 128 |
| Shared L2 Cache | 8MB, 8-way, 13 cycles, 64B |
| L2 MSHR Entries | 64 |
| Directory | Full-bit vector sharer list, 6-cycles |
| Interconnection Network | Grid, 64B links, 2-cycle link latency |
| Main Memory | 48GB/s bus, 105ns latency |
| TLS | |
| Cycles to Spawn | 20 |

Table 1: Architectural parameters.

Wattch [5]. We generate the binaries using a version of GCC 3.4.4 specially modified to operate with SESC. The TLS binaries are generated through automatic instrumentation using the Cetus source-to-source compiler [11]. Selection of the speculative regions was done through manual profiling. We note, however, that this was done in the interest of time and does not constitute a major limitation as existing automated TLS profiling frameworks – such as POSH [27], which was not used due to its inability to handle explicitly multithreaded applications – have been shown to be very effective.

3.2 System Models

The main microarchitectural and system features are listed in Table 1. The system we simulate is a many-core with 128 cores, where each core is a 4-issue out-of-order superscalar akin to an Intel Core 2 [19]. The on-chip network is hierarchical, where cores within a cluster are connected via a snooping bus and clusters are connected via a point-to-point network. Contention is fully modeled at all levels of the interconnect and at the shared L2 caches. Figure 4 depicts the topology of the chip.

We have extended SESC to support our hierarchical hybrid snooping-directory MESI invalidation protocol (Section 2.3.2) and our variant of TLS (Section 2.3.1). All coherence and TLS transactions are handled in detail both in terms of function and timing behavior.

For comparison purposes we also roughly evaluate two competing alternative systems: Core Fusion [20] and Frequency Boosting, inspired by [18]. Core Fusion dynamically combines the computational resources of several cores together to deal with lowly-threaded workloads, while separating the cores in case of highly-threaded ones. Core Fusion is approximated by modeling a many-core comprising of wide 8-issue cores with all the core resources doubled (L1 caches, ROB, Instruction Window, etc.) and without increasing the associated latencies. Thus, our model of Core Fusion is more aggressive than what can be implemented in practice and represents an *upper bound* of the performance that can be achieved with the technique. Frequency Boosting is modeled as follows: for each idle core one other core gains a frequency boost of 800MHz (we assume a 0.2V increase in core voltage that results in the same power cap). Note that this is a static scheme where half the cores are switched off for the entire execution of the application and the remaining half gain a constant boost in frequency. Intel’s

Algorithm 1 Hill-climbing approach for choosing the number of explicit and implicit threads.

```

1: for all omp parallel regions do
2:   if can_determine_max_iter_count_statically  $\wedge$ 
   (max_iter_count  $\leq$  128) then
3:     orig_cores  $\leftarrow$  cur_cores  $\leftarrow$   $2^{\lceil \lg(\text{max\_iter\_count}) \rceil}$ 
4:   else
5:     orig_cores  $\leftarrow$  cur_cores  $\leftarrow$  32
6:   end if
7:   evaluate cur_cores
8:   repeat //start searching downwards
9:     evaluate cur_cores  $\div$  2
10:  until detect_slowdown
11:  opt_cores  $\leftarrow$  cur_cores  $\cdot$  2
12:  if opt_cores = orig_cores then // search upwards as well
13:    repeat
14:      evaluate cur_cores  $\cdot$  2
15:    until detect_slowdown
16:    opt_cores  $\leftarrow$  cur_cores  $\div$  2
17:  end if
18:  if opt_cores < 64 then enable_TLS4
19:  else if opt_cores = 64 then enable_TLS2 end if
20: end for

```

Turbo Boost [18] is a dynamic scheme which is enforced at coarse-grain intervals when cores enter lower performance states. It is thus better applicable to multi-programmed workloads where cores are idle for long periods of time. In multithreaded workloads like the ones used in this study, however, Turbo Boost would not be triggered since all cores are active all the time. Our static Frequency Boosting policy is better suited for such scenarios. The shared portion of the memory subsystem, which includes the system bus, last-level shared cache, MSHRs, and on-chip interconnect, have the same parameters across the different configurations.

3.3 Benchmarks

Benchmarks from the PARSEC [3], SPLASH2 [43] and the OpenMP C version of the NAS NPB (v2.3) [1] suites are evaluated. From PARSEC *bodytrack* (OpenMP version), *canneal*, *streamcluster*, and *swaptions* are used. From SPLASH2 *cholesky*, *ocean-ncp*, *radiosity*, and *water-nsquared* are used. From the NPB, *ep*, *ft*, *it*, and *sp* are used. Some of the benchmarks from the aforementioned suites were not included in our evaluation because they either: (a) did not work with our infrastructure up to 128 threads (*freqmine*, *dedup*, *facesim*, *ferret*, *x264*, and *vips*), (b) scaled all the way to 128 threads for a very small data set (*blacksholes*, *fluidanimate*, and *lu_SPLASH2*), (c) showed similar scaling and bottlenecks as other benchmarks of the same suite (*barnes* and *radix* scaled similarly to *water*, *volrend* and *raytrace* similarly to *radiosity*, *fmm* scaled close to *ocean*, *mg*, *cg*, *bt*, and *lu_NAS* scaled similarly to *sp*), (d) were subsumed by benchmarks of another suite (*fft_SPLASH2* and *ft_NAS*). For each of the benchmarks, we simulate the parallel region to completion.

For TLS, speculative spawn and commit points were added to loops within hot functions. Compiler transformations to reduce data dependences, such as variable privatization, induction variable elimination, reduction variable expansion and min-max expansion are also performed automatically. We also exploit the commutativity of *rand* functions in order to remove data dependences between calls to such functions in *streamcluster*, *swaptions*, and *ep*. Furthermore, register spilling is done at task boundaries so that all inter-task register dependences are guaranteed to be communicated through memory. Detailed information about the benchmarks showing the speculation types applied, the coverage of the speculative regions, and the datasets used is shown in Table 2.

| Benchmark | Description | Input Sizes | | Coverage of Speculative Regions ^a | Types of Speculation |
|-------------------|------------------------------|-------------|-------------|--|----------------------|
| | | Normal | Large | | |
| PARSEC | | | | | |
| bodytrack | Computer Vision | sequenceB_1 | sequenceB_2 | 59% | LLS |
| canneal | Chip Design | 100000.nets | 200000.nets | 99% | LLS |
| streamcluster | Data Mining | 4K | 8K | 92% | LLS |
| swaptions | Financial Analysis | 16 | 32 | 80% | LLS,MLS |
| SPLASH2 | | | | | |
| cholesky | Sparse Matrix Multiplication | tk15 | tk29 | 81% | LLS |
| ocean | Ocean Current Simulation | 130 | 258 | 87% | LLS,MLS |
| radiosity | Graphics Rendering | test | room | 69% | LLS,MLS |
| water | Molecular Dynamics | 512 | 1000 | 99% | LLS |
| NAS OpenMP | | | | | |
| ep | Random Number Generator | 1M | 4M | 100% | LLS,MLS |
| ft | 3D FFT PDE | 128K | 512K | 42% | LLS |
| is | Integer Sort | 65K | 1M | 4% | LLS |
| sp | 3D Fluid Dynamics | 36 | 64 | 88% | LLS |

^a The coverage of speculative regions is reported for the sequential run.

Table 2: Simulated workloads. (LLS: Loop Level Speculation, MLS: Method Level Speculation).

4. EXPERIMENTAL RESULTS

4.1 Performance and Scalability

The top plots for each benchmark in Figure 6 show the performance of the different schemes as the number of cores is increased. The bottom plots show the breakdown of execution times according to busy versus synchronization (further divided into barrier and lock). The execution time bars are normalized to the baseline for the corresponding number of cores. For the baseline scheme the number of cores on the x-axes correspond to the number of explicit threads used. For Core Fusion and Frequency Boosting the number of cores correspond to the equivalent amount of resources used, which are twice the number of explicit threads used (i.e., Core Fusion merges two cores to run a single explicit thread and Frequency Boosting switches off one out of two cores to boost frequency of the active core). For our proposed scheme the number of cores on the x-axes correspond to the total number of explicit plus implicit threads used. Finally, for the baseline, Core Fusion, and Frequency Boosting schemes, in the experiments with fewer threads than the number of cores in the system (which is fixed at 128) we report results for the best mapping of threads to tiles. For our proposed scheme we always map one explicit thread and its associated implicit speculative threads to each tile.

4.1.1 Detailed Analysis

bodytrack (Figure 6a) exhibits poor scaling due to serial sections of code between parallel regions [3]. The increasing importance of those serial sections and the imposed load imbalance is reflected in the increased time spent in locks. By speculatively parallelizing loops in the serial sections as well as loops in the parallel sections we are able to improve its scalability. Speculated loops include loops in methods *ImageMeasurements::EdgeError()*, *ImageMeasurements::InsideError()*, *ProjecteCylinder::ImageProjection()*, *FlexFilterRowV()*, and *FlexFilterColumnV()*. The high iteration count of the loops that we speculate upon reflects in the large 4-way TLS performance improvement over 2-way. Core Fusion and Frequency Boosting also produce significant speedups over the baseline. This is due to faster execution of the serial sections. Core Fusion is more successful due to its ability to better exploit the relatively high instruction level parallelism (ILP) found in some of the hot loops of the benchmark.

canneal (Figure 6b) exhibits almost linear scaling up to 32 cores; going beyond that yields no further performance improvement. This is due to increased contention to the

lock in the *Rng* object constructor and increased barrier time as we increase the number of cores, and due to high cache miss ratio and relatively high inter-thread communication [2]. We speculatively parallelize the hot loop in the *annealer_thread::Run()* method which takes most of the program’s parallel execution time (Table 2) and are able to improve scaling to 64 cores. Despite the high-trip count of the hot loop and its high coverage, the TLS speedup, and especially the 4-way one, is only modest as the application becomes memory bound at this point. Core Fusion and Frequency Boosting offer minor performance improvements, again due to the memory-boundedness.

streamcluster (Figure 6c) exhibits good scaling up to 64 cores, but shows a slow down when going from 64 to 128 cores. As the number of cores is increased the amount of work done by each thread decreases, except for some constant work done by the master thread at the beginning of the main kernel and in some of the steps of the *pgain()* function. This incurs load imbalance and manifests as increased barrier time. We speculatively parallelize most of the loops in the *pgain()*, as well as the calls to *random()*, thus reducing load imbalance and improving scaling. Both Core Fusion and Frequency Boosting perform comparatively to TLS.

swaptions (Figure 6d) scales well with respect to number of *swaptions* in the input [3]. If the number of *swaptions* to be priced is less than the number of available cores the remaining cores are unutilized. The amount of computation done for each input *swaption* is constant and the loops are amenable to speculative parallelization and account for a significant portion of the program’s execution time. Moreover, the speculation coverage is further increased by exploiting the permutability of the function call *RanUnif()*, which accounts for 10% of the total execution time. This translates to substantial performance increase for the 2-way and 4-way TLS versions. Both Core Fusion and Frequency Boosting are able to attain increased performance through higher ILP, albeit significantly less than the TLS versions.

cholesky (Figure 6e) scales poorly and only up to 32 cores. This can be attributed to the high fraction of time spent on synchronization points, a relatively high cache miss rate, and high communication-to-computation ratio [43]. By speculatively parallelizing the loops in functions *ModifyTwoBySupernodeB()*, *ModifyBySupernodeB()*, *OneMatMat()*, *OneDiv()*, and *OneLower()* we are able to improve the application’s highest scalability point, albeit by a small margin. This is primarily due to a relatively high mispeculation rate, and secondarily due to being relatively memory bound. Core

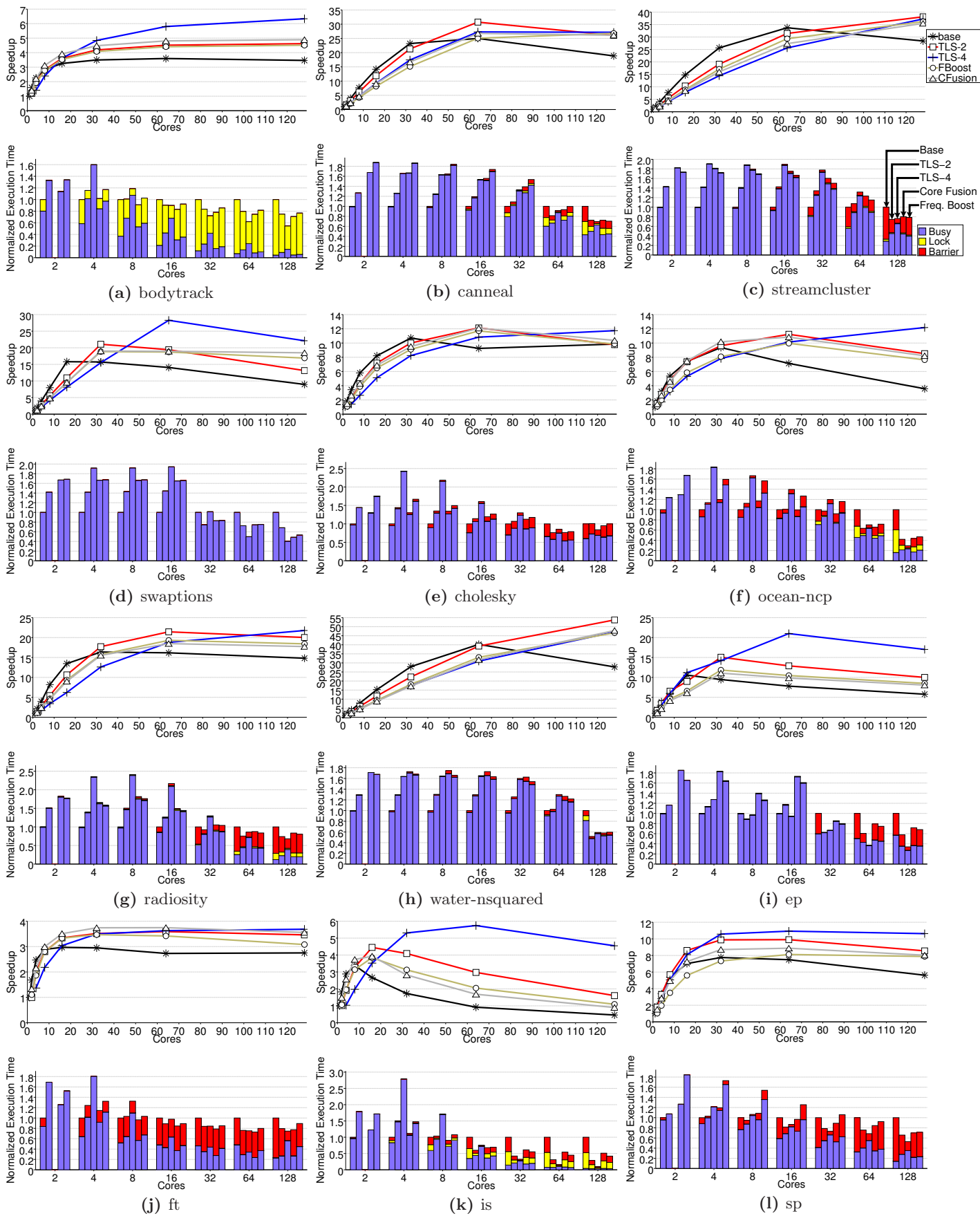


Figure 6: Performance, scalability and bottleneck breakdown.

Fusion and Frequency Boosting also offer marginal improvements to scalability.

ocean (Figure 6f) is characterized by intense usage of barriers and fine grain locks, as well as a relatively high cache miss ratio [43]. This limits its scalability and in fact causes a slow down when going from 32 to 64 cores. We speculatively parallelize most of the loops in the *slave2()* function, effectively overlapping calls to *laplacalc()* and *jacobcalc()*. Loops in the functions *relax()*, *rescal()*, and *intadd()* are also speculatively parallelized. This speeds up each of the computational steps and thus reduces the time spent in the barriers that succeed each step. Moreover, the well known prefetching side-effect of TLS [27] further improves performance by issuing misses early. Most loops speculated upon have a constant trip count of 2 which is reflected in the lack of performance improvement of the 4-way TLS version over 2-way. Core Fusion’s increased tolerance of long latency events enables significant performance improvements close to that of 2-way TLS. Frequency Boosting, on the other hand, increases the last-level cache miss latency in terms of core cycles and thus performs unfavorably to the other schemes.

radiosity (Figure 6g) uses tasks managed by distributed task queues. These tasks vary in size and, despite the use of task stealing, entail load imbalance beyond 16 cores, thus limiting the application’s scalability. This is evident from the increased time spent in the user defined synchronization (barrier) as the number of cores is increased. By speculatively parallelizing the hot loop in the *compute_visibility_values()* function in the heavy *visibility_task* and the calls to the *compute_form_factor()* function in the also heavy *ff_refinement_task* we effectively reduce load imbalance, thus allowing the application to scale better. Both Core Fusion and Frequency Boosting are able to attain increased performance through higher ILP, albeit significantly less than the TLS versions.

water (Figure 6h) scales very well up to 64 cores, but beyond this point there is a slowdown due to additional synchronization and communication. The hot loops in functions *INTERF()* and *POTENG()* that perform most of the computations for each molecule show infrequent data dependences and are amenable to speculative parallelization. We additionally parallelize speculatively loops in functions *INTRAF()*, *PREDIC()*, and *CORREC()* thus covering most of the application’s parallel execution (Table 2). This translates to a significant speedup of the 2-way TLS over the best performing baseline, scaling all the way to 128 cores. The 4-way TLS performs unfavorably compared to the 2-way TLS due to a large increase in the number of squashed threads. Core Fusion and Frequency Boosting perform almost identically in this benchmark, both failing to match the performance of the 2-way TLS.

ep (Figure 6i) scales relatively well up to 16 cores, after which there is a significant performance drop. The data partitioning is done statically, which leads to load balance problems. This is the case for the evaluated working set, which shows performance degradation that manifests as increased barrier time. The application has one hot loop which shows infrequent dependences and is amenable to speculative parallelization with synchronization around dependences. Moreover, we speculatively parallelize calls to *vranlc()* which precede this loop and account for the remaining execution time. This yields considerable performance improvement for the TLS versions. Both Core Fusion and Frequency Boosting provide performance benefits, but are not able to match that of our scheme.

ft (Figure 6j) is a memory bound application that scales poorly. The memory bandwidth quickly becomes the bottleneck as the number of cores is increased, and going beyond 8 cores offers no benefits. Speculatively parallelizing several loops helps improve scalability, but does not change the fact that the application is memory bound. Core Fusion performs relatively well by having a better long latency tolerance. Frequency Boosting, on the other hand, exacerbates the memory boundedness issue and performs worse than the other schemes.

is (Figure 6k) exhibits poor scaling due to a coarse-grain critical section. We speculatively parallelize the critical section as well as the kernel preamble which is executed only by the “master” (in *OpenMP* terminology) thread and is followed by a barrier. This provides an improvement in performance, enabling the application to continue scaling up to 16 cores for 2-way TLS and up to 64 cores for 4-way TLS. Both Core Fusion and Frequency Boosting also improve performance, but only up to 16, and less so than 2-way in this case.

sp (Figure 6l) exhibits poor scaling beyond 16 cores and none beyond 32 cores mostly due to synchronization time spent in barriers due to load imbalance, and to a lesser extent due to memory boundedness. Most of the loops of this benchmark are nested ones with depths ranging from two to four and work partitioning is done at the outermost level, yielding work only for the first *PROBLEM_SIZE* (36 for the evaluated dataset) threads. We speculatively parallelize loops in functions *add()*, *lhsy()*, *lshz()*, *compute_rhs()*, *x_solve()*, *y_solve()*, and *z_solve()*, consisting mostly of inner loops of the aforementioned outer loops. This reduces the execution time of the outer loops, which in turn translates to less time spent in barriers. Core Fusion is able to provide some performance improvement while Frequency Boosting fails to provide any noticeable benefits.

4.1.2 Summary

Overall, the results show that employing fine-grain implicit speculative threads to explicit threads beyond the scalability limit of the latter leads to performance gains in all the applications studied. In cases where there is enough fine-grain parallelism (e.g., loops with large trip counts) the 4-way TLS was able to improve on the 2-way TLS for even larger number of cores, after lagging behind for smaller number of cores. The alternative approaches of Core Fusion and Frequency Boosting are also often able to provide performance gains, albeit not in all cases and not as high as the TLS schemes. The difference between TLS and Core Fusion shows that exploiting additional fine-grain TLP can be advantageous as compared to exploiting more ILP, specially when enough parallelism can be harvested for the 4-way TLS while fusing 4 cores becomes impractical. Finally, the results show that Frequency Boosting can be effective on compute-bound applications, but is not as effective as the other approaches in memory-bound applications.

4.2 Effect of Dataset Sizes

As shown in the previous section, the technique proposed in this paper is effective in providing additional performance gains once applications stop benefiting from more explicit threads. So far we have assumed *strong*-like scaling by fixing the input dataset, i.e., the “Normal” dataset (see Table 2). A valid question is how the approach would fare under *weak*-like scaling conditions where the input of interest to the user is allowed to increase. To assess this we have also simulated the “Large” datasets, and present the results in Figure 7.

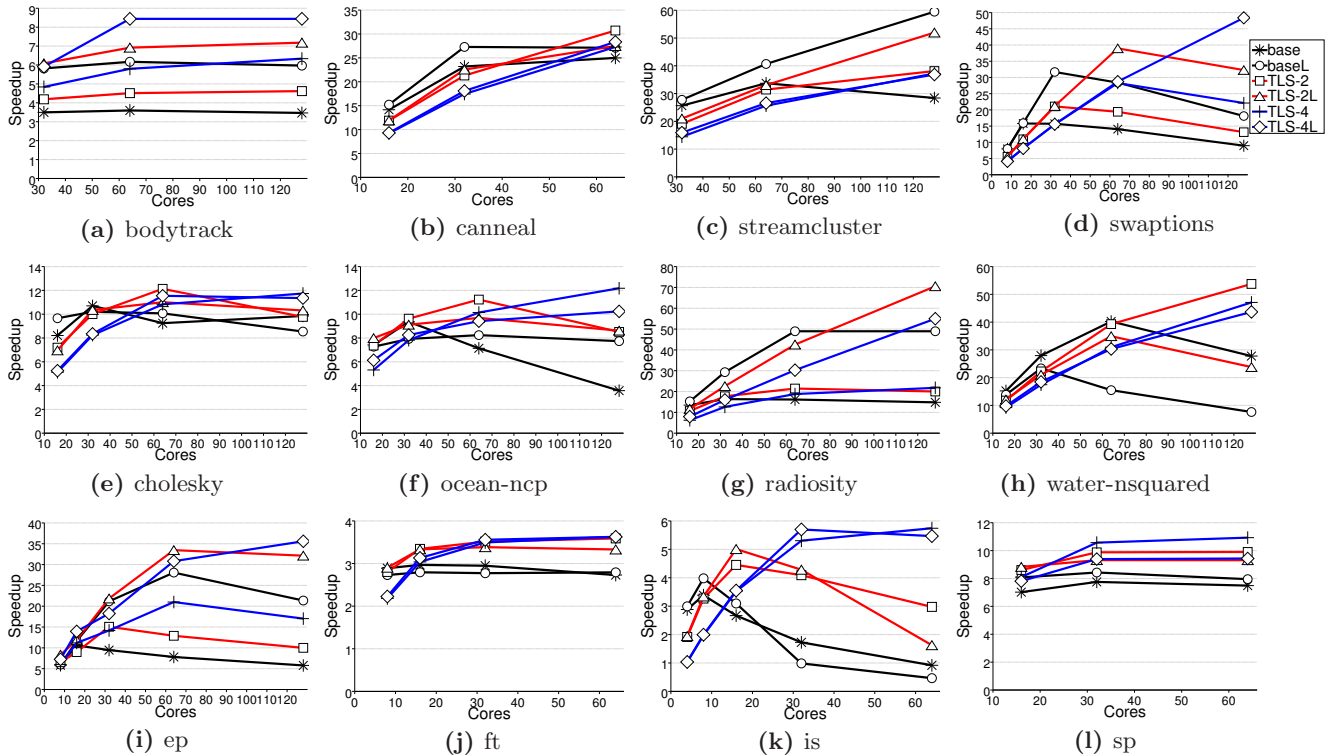


Figure 7: Performance and scalability of larger datasets.

Overall, as expected, most applications achieve performance improvements with a larger number of cores with the larger datasets, one of them (*streamcluster*) showing good scalability up to the total system size. However, in several cases performance still stops scaling before the total number of cores available is reached. In these cases, we again observe that our approach with 2-way and 4-way TLS is able to provide further performance improvement with larger numbers of cores.

4.3 Energy Consumption

Figure 8 depicts the total energy consumption of each of the evaluated schemes for all the benchmarks, normalized against the best performing baseline point. If we take *is*, for example, the base case is for 8 cores, the 2-way TLS, Core Fusion and Frequency Boosting are for 16 cores (and 8 explicit threads), and the 4-way TLS is for 64 cores (16 explicit threads). It is evident from this graph that Core Fusion’s excessive power consumption is not coupled with enough performance improvement to be a viable solution energy-wise for the applications studied. Frequency Boosting offers better energy efficiency than Core Fusion overall but is still 27% more energy hungry than the baseline on average.

Our scheme is able to achieve significant performance improvements while maintaining a reasonable power envelope that yields total energy consumption close to – and sometimes better than – the baseline. This is largely due to the fact that the speculative resources are only enabled within the speculated regions, in contrast to Core Fusion and Frequency Boosting which are enabled throughout. Our scheme is less energy efficient than the baseline in cases of high mis-speculation rates, like *cholesky*, or when it imposes increased contention of the shared resources, like *streamcluster* and, to a lesser extent, 4-way *ep*. However, for *bodytrack*, *canneal*, *ft*, *is*, *sp*, and to a lesser extent, *radiosity*, it is actually *more* energy efficient than the baseline. This is due to reduced time spent in the busy-wait synchronization primitives, which are particularly energy hungry.

4.4 Dynamically Tuning the Number of Threads

As explained in Section 2.4 we have developed an auto-tuning mechanism to dynamically choose the number of explicit and implicit threads for *OpenMP* applications. We have employed this to all the evaluated *OpenMP* benchmarks apart from *bodytrack*, whose parallel regions include code dependent on the thread identifiers and does not support dynamically switching the number of parallel threads. Figure 9 compares this auto-tuning mechanism against the static optimal extracted from the results of Section 4.1. Our auto-tuning mechanism performs within 10% of the optimal in the worst case (*is*), 1% in the best case (*ep*), and 6% on average. *ep*’s high iteration count completely amortizes the training costs. In the case of *is*, the auto-tuning settles correctly at 8 explicit threads, which is the tipping point of the baseline, but not the optimal in performance when coupled with 4-way TLS. *sp*’s parallel region is only encountered a few times and is thus more susceptible to training noise. *ft* also exhibits a low trip count parallel region, but shows less performance difference between adjacent thread counts. These results assert that the evaluated parallel applications retain their autotuning amenability even when we employ implicit threads. Furthermore, our proof-of-concept auto-tuning algorithm offers a feasible solution with performance benefits that are close to the static oracle.

5. RELATED WORK

TLS has been a topic of intense investigation over the years, but the vast majority of previous work has applied it to single-threaded applications. The implicit goal of such prior work on TLS, thus, was to achieve scalable parallel performance solely through implicit speculative threads, while our goal is to improve the scalability of explicitly parallel programs. TLS has been applied to individual threads of parallel applications in [42], but its overall impact on the entire application has not been considered; their focus lies in task selection and they only emulate TLS to estimate the overlap potential of each task. The possibility of applying

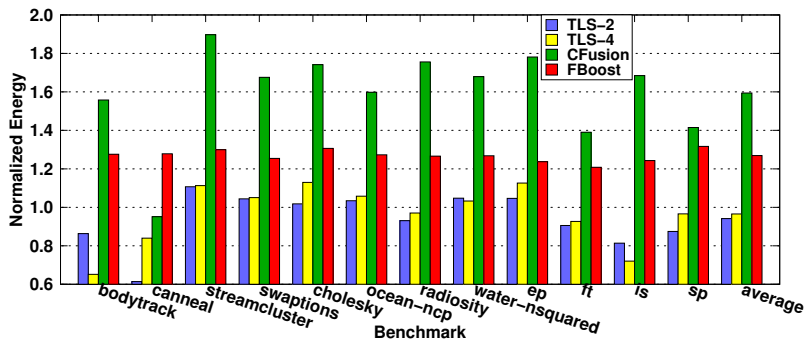


Figure 8: Energy consumption showing the best performing point for each scheme, normalized to the best performing base case.

TLS to improve the scalability of parallel applications has been briefly raised in recent studies on coping with Amdahl’s law in many-cores (e.g., [15]). However, these have not investigated the architecture implications or the trade-offs involved when combining explicit threads with TLS. To the best of our knowledge, this is the first work to investigate these trade-offs and propose a viable architecture for this purpose.

The closest prior work to ours is [21], where Speculative Parallel-stage Decoupled Software Pipelining (S-PS-DSWP) (originally proposed in [17]) is used to parallelize threads of a parallel application. Speculative Parallel-stage Decoupled Software Pipelining (S-PS-DSWP) [17] is a technique that supports speculative parallel execution of loop iterations, with nested sub-transactions within each iteration. Besides the differences between TLS and S-PS-DSWP as well as between the architectures, the main difference between that work and ours is that we investigate the trade-offs in scalability between employing more outer explicit threads versus more inner speculative implicit threads, and we also propose a mechanism for dynamically and automatically identifying the best trade-off.

Nested explicit and non-speculative parallelism is commonly exploited in the high-performance community through MPI-OpenMP environments. Also, nested explicit speculative parallelism has been proposed with nested transactional memory (e.g., [32]). Unlike our implicit speculative TLS threads, nested transactional memory is not transparent to the user and requires additional programming effort.

Two prior works [33, 34] have considered architectures that support both explicit and TLS threads. They propose the use of explicit parallelism where available, and the use of TLS only outside these explicit parallel regions. Thus they switch between explicit and implicit thread modes, and do not support both types of threads simultaneously in a nested fashion.

In addition to works that consider nested and speculative parallelism, our work is also very much related to current efforts that attempt to mitigate Amdahl’s law through other dynamic and transparent means. Core-fusion [20] does so by dynamically “merging” cores together and exploiting ILP when sufficient explicit parallelism is not available. Recent commercial multi-cores incorporate frequency boosting [18] to shift resources to a subset of cores in order to improve performance when the workload does not provide enough parallelism to utilize all cores. The work in [28] proposed a hardware/software scheme to improve performance of sequential applications using speculative fine-grain multithreading. It uses a clustered architecture, similarly to this work, but only evaluates single threaded applications. The now canceled ROCK architecture [8] proposed the use of automatic, hard-

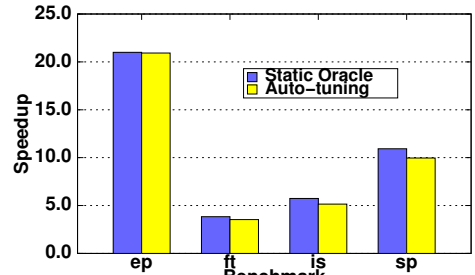


Figure 9: Dynamically tuning the number of threads.

ware implicit threads to complement sequential execution by either exposing more ILP or MLP.

Speculative Lock Elision (SLE) is another similar approach [30, 35, 36], that allows explicit threads to run speculatively ahead of a synchronizing operation. SLE tries to overlap some of the work inside critical sections with work being done by other threads outside the critical section, while our approach is to accelerate the execution of the critical section itself. Like our approach, this technique attempts to reduce the amount of time lost in synchronization in the case of large critical sections and load imbalance (Figure 2a,c). It offers little benefits, however, for applications limited by regions non proportional to the dataset (Figure 2b). Also, unlike our nested approach, SLE requires speculation control across the entire system, which makes the scheme harder to scale to many-core systems.

Finally, [40] speeds up applications with significant critical sections using a static heterogeneous architecture. Our approach, and those mentioned above, provide a more dynamic approach to deal with critical sections and Amdahl’s law.

6. CONCLUSION

With the advent of multi-cores, programmers have to endorse parallel programming if they are to exploit the additional hardware resources to improve their applications’ performance. Fine-grain parallelism is hard and error-prone, however, and programmers usually avoid parallelizing their applications using fine-grain threading. They instead focus on uncovering parallelism at a coarser granularity which offers a good compromise between performance improvement and development time.

In this paper we have proposed using implicit speculative parallelism to complement user-level explicit threads. Our scheme is able to improve a parallel application’s performance beyond its higher scalability point by using cores that would otherwise be underutilized to run implicit speculative threads. Experimental results on a simulated 128-core show that performance improves on top of the highest scalability point by as much as 98%, and 41% on average, for the 4-way TLS and by as much as 42%, and 27% on average, for the 2-way TLS. Finally, we present an auto-tuning mechanism to dynamically choose the number of explicit and implicit threads for *OpenMP* applications which performs within 6% of the static oracle thread allocation.

7. REFERENCES

- [1] D. H. Bailey et al. The nas parallel benchmarks – summary and preliminary results. In *SC*, December 1991.

- [2] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding parsec performance on contemporary cmps. In *IISWC*, October 2009.
- [3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.
- [4] C. Blundell et al. Deconstructing transactional semantics: The subtleties of atomicity. In *WDDD*, June 2005.
- [5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, June 2000.
- [6] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, June 2006.
- [7] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA*, February 2007.
- [8] S. Chaudhry et al. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's ROCK processor. In *ISCA*, June 2009.
- [9] M. Cintra, J. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA*, June 2000.
- [10] M. Curtis-Maury et al. Prediction models for multi-dimensional power-performance optimization on many cores. In *PACT*, October 2008.
- [11] C. Dave et al. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, December 2009.
- [12] B. R. de Supinski. *Personal Communication*. Lawrence Livermore National Laboratory, May 2011.
- [13] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *ISCA*, June 2010.
- [14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, October 1998.
- [15] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, July 2008.
- [16] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.
- [17] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *CGO*, April 2010.
- [18] Intel. *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*, White Paper, November, 2008.
- [19] Intel Corporation. *Intel Core2 Duo Processors and Intel Core2 Extreme Processors for Platforms Based on Mobile Intel 965 Express Chipset Family Datasheet*, 2008.
- [20] E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: Accommodating software diversity in chip multiprocessors. In *ISCA*, 2007.
- [21] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *MICRO*, December 2010.
- [22] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS*, July 1998.
- [23] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *ISCA*, June 2005.
- [24] K. Kusano et al. Performance evaluation of the omni openmp compiler. In *ISHPC*, October 2000.
- [25] E. A. Lee. The problem with threads. *IEEE Computer*, January 2006.
- [26] D. Lenoski, J. Laudon, K. Guarachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *ISCA*, May 1990.
- [27] W. Liu et al. POSH: a TLS compiler that exploits program structure. In *PPoPP*, March 2006.
- [28] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. Boosting single-thread performance in multi-core systems through fine-grain multi-threading. In *ISCA*, June 2009.
- [29] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS*, June 1999.
- [30] J. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, October 2002.
- [31] C. C. Minh et al. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, June 2007.
- [32] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS*, October 2006.
- [33] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *ICS*, 2001.
- [34] L. Porter, B. Choi, and D. M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *PACT*, September 2009.
- [35] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, December 2001.
- [36] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.
- [37] J. Renau et al. SESC simulator, January 2005. <http://sesc.sourceforge.net>.
- [38] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS*, June 2005.
- [39] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA*, June 1995.
- [40] M. A. Suleman et al. An asymmetric architecture for accelerating critical sections. In *ASPLOS*, 2008.
- [41] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical report, Compaq Research Lab., 2006.
- [42] C. von Praun et al. Implicit parallelism with ordered transactions. In *PPoPP*, March 2007.
- [43] S. Woo et al. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, June 1995.
- [44] L. Yen et al. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA*, June 2007.