Automatic SMT Threading for OpenMP Applications on the Intel Xeon Phi Co-processor

Wim Heirman[1,2] Trevor E. Carlson[1] Kenzo Van Craeynest[1]
Ibrahim Hur[2] Aamer Jaleel[3] Lieven Eeckhout[1]

[1] *Ghent University, Belgium*
[2] *Intel, ExaScience Lab* [3] *Intel, VSSAD*

ABSTRACT

Simultaneous multithreading is a technique that can improve performance when running parallel applications on the Intel Xeon Phi co-processor. Selecting the most efficient thread count is however non-trivial, as the potential increase in efficiency has to be balanced against other, potentially negative factors such as inter-thread competition for cache capacity and increased synchronization overheads.

In this paper, we extend CRUST (ClusteR-aware Undersubscribed Scheduling of Threads), a technique for finding the optimum thread count of OpenMP applications running on clustered cache architectures, to take the behavior of simultaneous multithreading on the Xeon Phi into account. CRUST can automatically find the optimum thread count at sub-application granularity by exploiting application phase behavior at OpenMP parallel section boundaries, and uses hardware performance counter information to gain insight into the application's behavior. We implement a CRUST prototype inside the Intel OpenMP runtime library and show its efficiency running on real Xeon Phi hardware.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Threads*; C.5.1 [**Computer System Implementation**]: Large and Medium ("Mainframe") Computers—*Super (very large) computers*

Keywords

Simultaneous multithreading, OpenMP, auto-tuning

1. INTRODUCTION

Modern processor architectures support SMT, or multiple hardware threads per core. Running more than one thread per core can improve utilization of the core's execution resources, and help in hiding memory access latencies [6, 9]. However, increasing the application's thread count can also lead to increases in synchronization overhead and load imbalance, or inflate per-core working set sizes which (as threads share TLB and cache capacity, among other resources) can transform well-behaving cache-fitting applications into cache-trashing ones making their performance limited by off-chip bandwidth [4]. Selecting the best per-core thread count is therefore an important optimization problem. Architectures such as the Intel Xeon Phi (codenamed Knights Corner), which support running up to four threads on each of its 60+ cores, are especially susceptible to this behavior.

Common practice is for application designers to test performance of their application using a range of thread counts, and make recommendations to users as to which thread count works best. Yet, as we will show in Section 2, applications do not necessarily have a single optimum thread count, as this optimum can shift depending on the input set used. Also, for more complex applications that exhibit phase behavior, a thread count that is optimum for one phase may not necessarily be the right choice for other phases.

Curtis-Maury et al. propose dynamic concurrency throttling (DCT), an automated way of running the optimum thread count – which is often lower than the number of cores available in the system – at each point during the application's execution [2]. The DCT methodology collects a number of performance counters for each application phase in a limited number of configurations (consisting of thread counts at various hierarchical levels, e.g., active cores and SMT threads per core), and uses a hardware-specific prediction model to extrapolate application performance for each configuration. DCT allows the optimum thread count to be selected for each application phase, using OpenMP program semantics to extract phase behavior (each `omp parallel` statement is considered as a distinct phase). Heirman et al. propose ClusteR-aware Undersubscribed Scheduling of Threads (CRUST), a variation on DCT specialized for clustered last-level cache architectures; and argue that clustered caches in combination with DCT-like techniques make the performance of future many-core designs more resilient to variations in application behavior [4]. In contrast to DCT, CRUST avoids using prediction models and uses direct measurement of application performance.

In this paper, we extend CRUST to incorporate the effects of simultaneous multithreading, which in addition to competition for cache capacity, exhibits additional effects incurred by core resource sharing. We implement this improved version of CRUST inside the Intel OpenMP runtime library [1] and explore its performance when running on Xeon Phi hardware.

ROSS '14, June 10, 2014, Munich, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2950-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612262.2612268

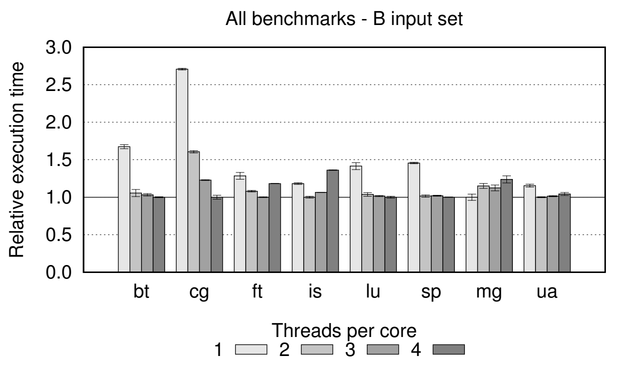

Figure 1: Application performance as a function of per-core thread count, normalized to the optimum thread count for each application.

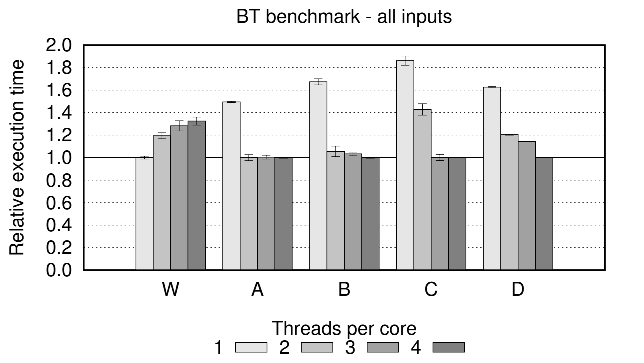

Figure 2: Performance of `bt` with different inputs, as a function of per-core thread count, normalized to the optimum thread count for each input set.

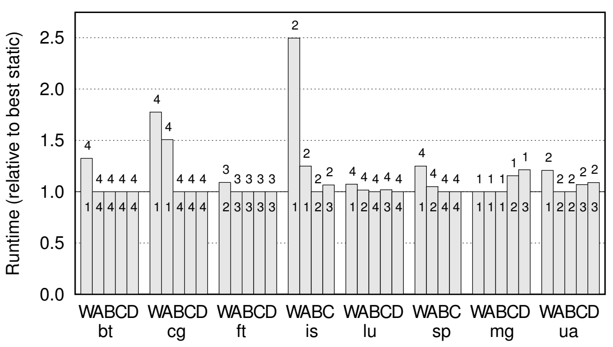

Figure 3: Application performance, relative to the best thread count, when optimizing each application using a single input set (input B).

2. MOTIVATION

We first explore the effect of various *static* per-core thread counts, i.e., we run the complete application with a fixed thread count set at startup using the `OMP_NUM_THREADS` environment variable. Each application is run with 60, 120, 180 or 240 threads, corresponding to 1...4 threads on each of 60 cores. `KMP_AFFINITY=scatter` was used to distribute threads evenly over cores. We run the NAS Parallel Benchmarks [7] with varying number of SMT threads on an Intel Xeon Phi 7120A system (see Section 5 for more details on the methodology used). Figure 1 shows the resulting execution time for all benchmarks from the NAS suite with their B input set. Results are normalized to the optimum thread count for each application and input set combination.

Performance clearly vary across benchmarks. Most applications run best when at least two SMT threads are used. For some benchmarks (`ft` and `is`), performance degrades when too many threads (three or four) are spawned, while in one case (`mg`), running just a single SMT thread is the better option by some margin.

An important, but often overlooked determining factor in this is the working set of the application. In Figure 2, we single out the `bt` benchmark and run it with five different input sets of increasing size. In this application, a larger input set also increases the per-thread working set. Input sets A and B exhibit the behavior often seen on Xeon Phi machines: a single thread has low performance because of the core's front-end restrictions (each thread can dispatch two instructions but only every other cycle) while two threads suffice to saturate the core. Three or more threads do not provide any benefit over two SMT threads per core for this workload. Moving to larger input sets (C and D), the application becomes significantly DRAM latency bound. Here, extra threads beyond two per core do improve performance as this increases the number of outstanding memory requests, which allows more of the DRAM latency to be hidden. The smallest input set W, however, exhibit the opposite behavior where low thread counts perform better. Here, the working set is small enough to fit into the core's private caches if only a single thread is run per core. With two or more threads per core, the threads compete for cache capacity. This inflates the cache miss rate and leads to an increase in off-chip memory accesses, and performance suffers significantly.

The optimum thread count thus depends on the application and the input set size. The application-dependent part is known by most users, who determine a per-application optimum thread count and record this into the application's best practices; however, this is often done using a single input set only. Figure 3 shows the performance of this optimization method. We determine the best thread count for each application using the middle B input set, then run the program with all other input sets using the thread count that performed best for input B. All results are again normalized to the optimum thread count for each benchmark and input combination. Numbers inside the graph denote the per-core thread count that was used, in the optimum case (numbers inside the bars, all corresponding to a relative execution time of 1.0) and for the case optimized using the B input (numbers above the bars). Clearly, many applications perform significantly worse when running them with a thread count that was optimized for a different input set.

In addition to input size dependence, real applications are often a complex compositions of multiple compute kernels, each of which may have its own distinct working set size and hence optimum thread count. The dynamic concurrency throttling methodology therefore argues for an *automated approach* of determining thread count *at sub-application granularity* [2]. In the next section, we propose a simple DCT approach that is geared at the Intel Xeon Phi and show how it can be easily integrated into the Intel OpenMP runtime.

3. AUTOMATIC OPENMP THREADING

DCT exploits application phases, and assumes OpenMP parallel sections are a good indicator of phase behavior. Indeed, each parallel loop signifies a distinct part of the source code, with its own instruction mix and working set size — both of which are the most important properties that affect execution resource and cache capacity sharing, and hence optimum thread count. DCT will find the optimum thread count for each parallel section, and execute an `omp_set_num_threads` call to update the thread count at the start of each section. This automatically triggers the OpenMP runtime to change data partitioning accordingly.

The CRUST methodology from [4] assumed a clustered cache hierarchy, where a small number of cores share a last-level cache. This restricts thread interaction to capacity sharing in the last-level cache. When applying this method to SMT threads sharing all of the core's execution resources, the sharing behavior becomes more complex, invalidating some of CRUST's assumptions. Its basic premise, which is to find per-section optimum thread counts while being cognisant of cache miss rates, remains valid. Concretely, we will use the CRUST-descend variant, but remove its early-exit clause. We also add aggregation of small sections, a suggestion made but not fully evaluated by [4]. Compared to the general DCT approach from [2], CRUST is much more straightforward to implement in that no hardware-specific model has to be constructed. In addition, CRUST can be implemented using just two hardware performance counters (which is the maximum number of counters that can be reliably instantiated on the Intel Xeon Phi), whereas the original DCT methodology requires a much larger number of performance counters to feed its performance models.

3.1 Per-section tuning

The CRUST algorithm treats each OpenMP parallel section individually, and goes through a number of phases as a given section occurs during execution of the application.

The first occurrence of each section is ignored. In many cases, subsequent occurrences of a single section operate on the same data. This makes the first occurrence different in that it potentially incurs many more (cold) cache misses. By ignoring the first occurrence we effectively allow for cache warmup before measuring a section's performance.

Starting with the second occurrence, CRUST enters its calibration phase. Occurrences two through five are run with different per-core thread counts, starting with four for the second occurrence to using just a single thread per core for the fifth. During each section occurrence, hardware performance counters are used to measure instruction count and second-level cache miss rates. In addition, the `rdtsc` instruction is used to measure elapsed time (in clock cycles). This allows CRUST to compute IPC (instructions per cycle) and L2 MPKI (L2 read misses per 1,000 instructions) as performance metrics corresponding to each per-core thread count option.

Once all options have been profiled, the best-performing one is selected for use during the remainder of the application. Determining the 'best' option is not straightforward, however. If the amount of work, and the number of instructions needed to perform that work, were equal in all occurrences of the section used in calibration, the best-performing option would be that with the lowest cycle count, or the highest IPC (both metrics would be equivalent as they are related through a constant factor, instruction count). However, program semantics can be such that the amount of work varies across occurrences of the same section, while spin loops can inflate instruction count even when the amount of work is constant. The original DCT approach of Curtis-Maury [2] uses an architecture-specific model to reconstruct useful performance based on a variety of hardware performance counters, hence eliminating the spinloop problem. However, the Xeon Phi can only support a limited number of simultaneously active performance counters, which are potentially shared with other auto-tuning or application analysis libraries. Models that require a plethora of performance counter inputs are therefore not feasible for implementation in production OpenMP libraries. In CRUST, we simplify the implementation by assuming that spin loops are the more common problem, and choose to pick the thread count that minimizes cycle count.

As a guard against changes in application behavior (such as the amount of work per parallel section), CRUST keeps monitoring instruction count and cache miss rates even after the calibration phase. At the end of each occurrence, these values are compared to those values recorded during calibration. If either the instruction count or the MPKI value change significantly, we assume a change in workload behavior has occurred and the section is recalibrated. We used a 30% relative change as the threshold, in addition to an MPKI of at least 3 misses per 1,000 instructions to avoid recalibration when miss rates are insignificantly low.

3.2 Aggregating small sections

While most applications use large parallel sections, some employ fine-grained parallelism leading to parallel sections of just a few million instructions long. Here, there is often data sharing *across* parallel sections. Good cache performance therefore requires that the data partitioning is not changed from one section to another. Hence, thread count should stay constant as well, which requires that small sections are tuned together rather than in isolation.

In CRUST, each OpenMP section starts off as being large, and is therefore individually tuned. At the end of each section, its runtime is compared to a threshold value. Shorter sections are marked as being small and will, the next time they are encountered, be aggregated into larger chunks of at least 50 million cycles each. The *aggregated* section is seen as one additional section, in addition to all *large* sections, and is tuned in the same way. For the aggregated section, maximum IPC is used rather than minimum cycle count, as the amount of work in each occurrence can be different when it consists of differing collections of small sections.

The optimum threshold to separate large from small regions was empirically determined at 50 million clock cycles (around 40.4 ms at our Xeon Phi's 1.238 GHz clock). This value is of the same order of magnitude as the average lifetime of data in the L2 caches. Indeed, sections smaller than this value cannot warm up the cache with their own data and rely on data reuse across sections — requiring a constant thread count and hence data partitioning. For larger sections, cache warmup effects at the beginning of the section can be amortized and it is more beneficial to tune the section individually.

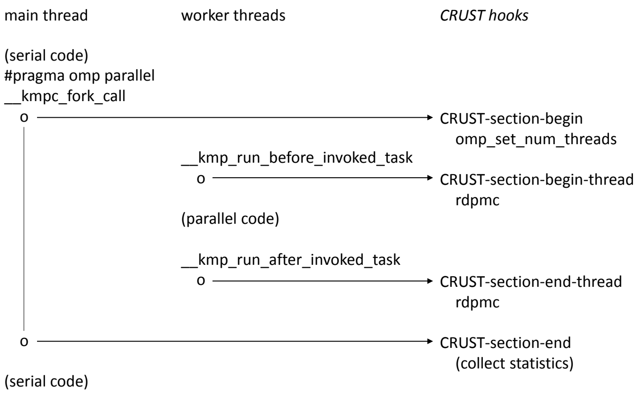

Figure 4: Flow of CRUST hooks and their triggers.

4. IMPLEMENTATION

4.1 OpenMP runtime library integration

We integrated CRUST into the open source version of the Intel OpenMP runtime library [1]. CRUST itself is a separate, reusable library, while the Intel OpenMP library was extended with a number of hooks that trigger CRUST functionality. This makes it easy to port CRUST to other OpenMP runtime libraries, or even non-OpenMP parallel runtimes such as Intel Threading Building Blocks (TBB) or Cilk Plus. By keeping all changes confined to the runtime, application changes are not needed — they can simply link dynamically to a CRUST-enabled runtime. An overview of the various hooks that implement CRUST, their main functionality and how they are integrated into the Intel OpenMP runtime library, can be found in Figure 4.

Inside of Intel's OpenMP library, the main functionality of OpenMP sections is implemented by `__kmpc_fork_call` which is called once for each `#pragma omp parallel` in the application source code. We added calls to CRUST's begin and end section hooks at the top and bottom of this function. Its `loc` argument uniquely identifies the current parallel section allowing CRUST to tune each section individually. In the begin section hook, the algorithm determines the number of threads that are to be used for the upcoming occurrence of that section. This will be either the next thread count to try in the calibration phase, or the optimum thread count for that section as previously determined. CRUST then calls `omp_set_num_threads` to update the active thread count accordingly. The end section hook collects performance metrics for the section that just executed, and decides when to stop aggregation in case small sections are encountered.

For all experiments, each thread is pinned to a private hardware context. We set `KMP_AFFINITY=scatter` so threads 0,1,…59 end up at the first SMT thread of cores 0…59; threads 60…119 are pinned to the second SMT thread of cores 0…59, etc. This way, setting the number of OpenMP threads to 60, 120, 180 or 240 always utilizes exactly 60 cores and runs between one and four SMT threads on each of them. The 61st core of the chip is left free for the OS.

4.2 Performance counters

The Xeon Phi processor includes a number of hardware performance counters, which can be accessed through the usual *perf* infrastructure available in recent Linux kernels. These are used by CRUST to obtain per-thread instruction counts and L2 miss rates. However, reading performance counters for all cores through the default kernel interface requires making a system call which takes millions of clock cycles to complete. Since CRUST requires performance counter information at the start and end of each (potentially small) parallel section, the overhead of using *perf* directly is too substantial.

Instead, we employ the `rdpmc` instruction directly, which reads one of the performance counters on the local core. This instruction can be executed from userspace once the proper setup is done.[1] At the start of the application, we use the regular *perf* interface to initialize the processor's performance counter infrastructure to record instruction count (`INSTRUCTIONS_EXECUTED` event) and the number of L2 cache misses (`L2_READ_MISS`). We then add hooks from inside the `__kmp_run_{before,after}_invoked_task` functions of the OpenMP runtime, which are executed by all threads at the start and end of a parallel section. Each executes the `rdpmc` instruction to read the hardware performance counters of the local core. Results are stored in a global data structure that can be read by CRUST's optimization algorithm running in the main thread.

Intel Xeon Phi	7120A
Active cores	61
Clock frequency	1.238 GHz
On-board memory	16 GB
ECC	enabled
Kernel version	2.6.38.8+mpss3.1.2
MPSS Version	3.1.2-1

Table 1: Hardware characteristics.

5. RESULTS

We now evaluate the performance of CRUST by running the NAS Parallel Benchmarks [7] with a CRUST-enabled OpenMP runtime on an Intel Xeon Phi 7120A system (see Table 1 for its main features). All benchmarks are run in native mode, i.e., they execute directly on the Xeon Phi co-processor without interaction with the host processor.

5.1 CRUST performance

Figure 5 compares the execution time of CRUST-enabled applications with both the best (fastest) and worst (slowest) static per-core thread count. Results are normalized to the fastest (static) case. For comparison, the worst static case is also plotted (dark bars). The numbers inside the bars denote the static per-core thread count for both the fastest and slowest options.

In most cases (except for `ft`), the best (static) case for the smallest input set (W) is to run a single thread per core, whereas utilizing all SMT contexts (four threads per core) is consistently the worst option. For the largest input set D, however, this situation reverses and one thread per core is

[1]Access to the `rdpmc` instruction from userspace is regulated by the processor's `CR4.PCE` configuration bit and is disabled by default, we implemented a small kernel module to enable this. Performance counter initialization uses the existing *perf* interface and requires no further kernel modifications.

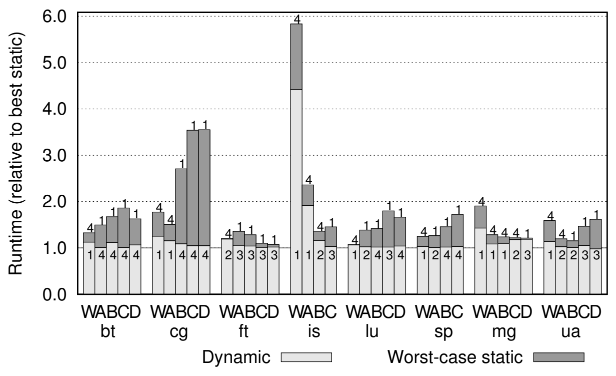

Figure 5: Relative execution time of CRUST-enabled applications relative to their best static thread count.

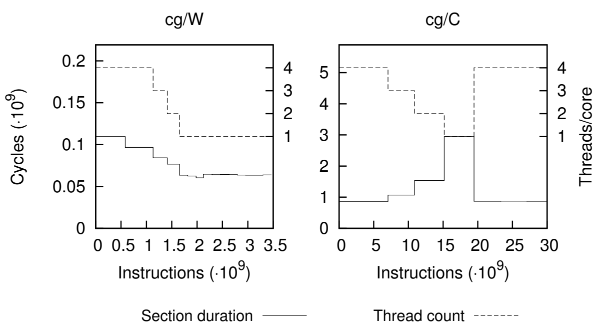

Figure 6: Through-time behavior of `cg`, W and C inputs.

always the slowest option while either three or four threads are needed to obtain the best performance.

CRUST, our dynamic technique, is able to obtain an execution time that is close to the optimum for almost all benchmark/input set combinations. The largest deviations occur for the smallest W input set, which for most benchmarks does not have enough iterations to amortize the calibration overhead.

5.2 Through-time behavior

In Figure 6, the through-time behavior of the CRUST algorithm can be observed, for the `cg` benchmark with two of its input sets (the smaller W input set on the left, and the much larger C set on the right). `cg` has just a single significant OpenMP section. Its first occurrence is run with the maximum thread count, CRUST ignores its results to make sure cache warmup effects do not distort calibration. Next, CRUST successively tries out running this section with four, three, two and one threads per core. Out of these four options, one thread per core is best (i.e., lowest cycle count) for the W input while four threads per core perform best for the C input set. This optimum thread count is used for the remainder of the application.

Figure 7 plots the through-time behavior of a more complex benchmark. `CoMD` from the Matevo suite [5] is a molecular dynamics simulation mini-application. Most of its time is spent computing interatomic forces which is a compute-bound problem that scales relatively well to multiple cores.

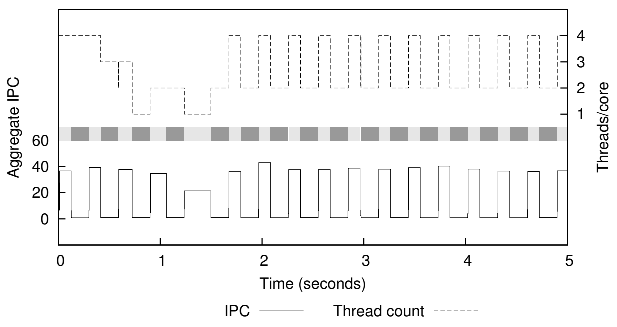

Figure 7: Through-time behavior of `CoMD`.

This phase consists of a single OpenMP parallel section, and is denoted by light grey boxes in Figure 7. The darker boxes consist of multiple serial and parallel phases, each shorter than 50 million clock cycles, and are therefore aggregated by CRUST and tuned as a single monolithic phase. After calibration, the light grey phase is selected to use four threads per core, which is the option that was found to minimize cycle count (and on this graph, maximize aggregate IPC) in the calibration phase. The other phase is mostly memory bound, here two threads per core are selected. For this benchmark, calibration ends after 1.6 seconds of runtime, the complete benchmark runs for 29 seconds (using a $40 \times 40 \times 40$ input domain). Using CRUST even provides a slight (2%) reduction in execution time over the fastest static case (which uses three threads per core for all sections), while the slowest case (one thread per core) increases execution time by 43%.

5.3 Power and energy consumption

To determine power consumption and possible energy savings resulting from CRUST, we measured instantaneous power consumption of the Xeon Phi co-processor while running the CRUST-enabled benchmarks. This was done by periodically reading from `/sys/class/micras/power` in a background thread. We then compute average power and cumulative energy consumption for the duration of each benchmark, and plot the results in Figure 8. Note that CRUST itself currently does not use power values in its algorithms, although an adaptation of the algorithm that is tuned to optimize energy savings rather than performance can be envisioned.

The slowest static option activates fewer core resources, and usually has a lower power draw associated with it as well. However, since the execution time is (often much) longer, the slowest option is the least energy-efficient one. Since CRUST is able to find the best thread count in most cases, its power and energy consumption are close to that of the fastest static option as well.

6. EXTENSIONS AND FUTURE WORK

Nested parallelism. We currently do not invoke any of the CRUST routines in nested parallel sections. While use of nested parallelism is therefore allowed when running CRUST, only the boundaries of the outermost level of parallelism are considered to represent phase behavior. A more detailed analysis of parallelism at all nesting levels may be able to expose more detailed phase behavior, and hence lead to

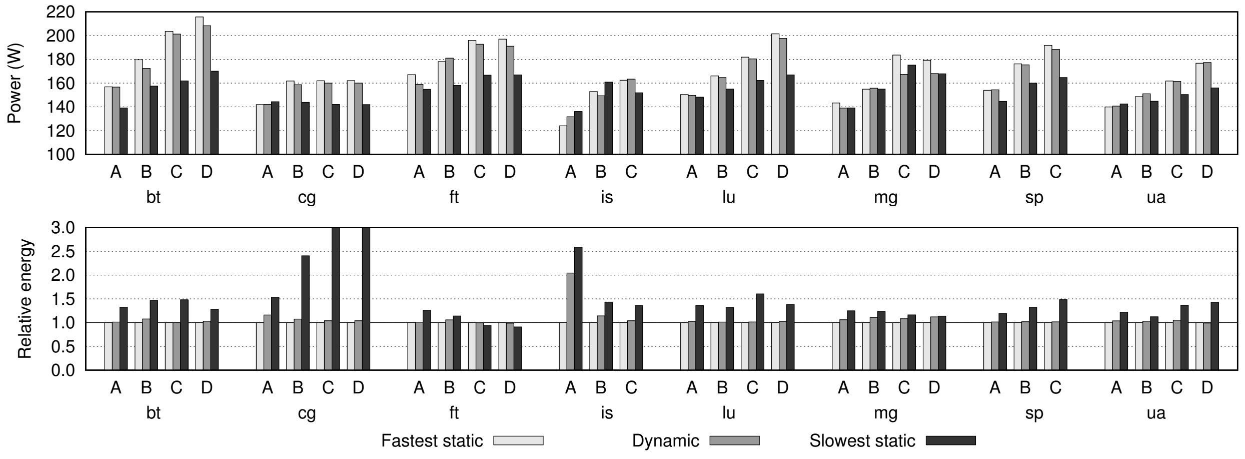

Figure 8: Power (top) and energy consumption (bottom) of the Xeon Phi co-processor while running the NAS benchmarks.

finer-grained thread count adaptation and potentially more efficient execution.

Undersubscription. In this work, we always utilize 60 cores of our 61-core Xeon Phi co-processor. Some applications, or some phases inside them, do not scale to this large number of cores. One extension of CRUST could be to, in addition to varying the number of threads per core, also change the number of active cores. This can lead to reductions in synchronization overheads, or for bandwidth-bound applications, allow cores to be disabled which saves energy while not affecting performance. To this end, CRUST could be combined with Synchronization-Aware Threading and/or Bandwidth-Aware Threading [8] (which could piggy-back on the already available L2 miss rate to estimate DRAM bandwidth). Alternatively, a multi-dimensional DCT model could be used — assuming a sufficiently accurate prediction model can be constructed based on the limited set of available hardware performance counters.

Recommendations for future hardware. As this paper and many others show, hardware performance counters are an extremely useful way of finding execution bottlenecks at runtime and adapting application behavior to avoid them. However, several restrictions on how performance counters can be accessed on current architectures make their use in runtime libraries cumbersome.

Only a small number of performance counters (just two in the Intel Xeon Phi) can be active simultaneously. This restricts the axes along which the application can gain visibility into its behavior, or requires the use of sampling (which has its own set of trade-offs). It also reduces orthogonality of adaptive runtime techniques, and requires that different runtime libraries that each may want to perform different types of optimizations need to coordinate to share performance counter resources. Adding a larger set of fixed-function, always-available counters that at least capture high-level application characteristics can help in this respect.

In addition, collecting performance counter information should be low-overhead and user-space only, encouraging fine-grained measurements and allowing small code regions such as spinlocks to be excluded. Finally, performance counters that indicate a direct impact on execution time, such as the top-down approaches presented in [3] and [10], are more useful than pure event counts or miss rates as the latter require extra modeling steps and (potentially invalid) assumptions to translate their observation into achievable performance gains.

7. CONCLUSIONS

We explored the performance of using different per-core thread counts on an Intel Xeon Phi system, and showed how the optimum thread count varies across applications, when changing the input set of some applications, and even within a single application when it is composed of multiple kernels. We then proposed our SMT-aware extension to CRUST, a methodology based on dynamic concurrency throttling which can determine the optimum thread count automatically. CRUST can be integrated easily into the OpenMP runtime library; by combining application phase behavior and leveraging hardware performance counter information it is able to reach the best static thread count for most applications — and can even outperform static tuning on more complex applications where the optimum thread count varies throughout the application.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported by Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium.

9. REFERENCES

[1] Intel OpenMP runtime library. Available at `http://www.openmprtl.org/`.
[2] M. Curtis-Maury, F. Blagojevic, C. Antonopoulos, and D. Nikolopoulos. Prediction-based power-performance

adaptation of multithreaded scientific codes. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1396–1410, Oct. 2008.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A top-down approach to architecting CPI component performance counters. *Micro, IEEE*, 27(1):84–93, 2007.

[4] W. Heirman, T. E. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout. Undersubscribed threading on clustered cache architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[5] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[6] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 136–145. ACM, 1992.

[7] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and its performance. Technical report, NASA Ames Research Center, Oct. 1999.

[8] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, 2008.

[9] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[10] A. Yasin. A top-down method for performance analysis and counters architecture. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.