# The Benefit of SMT in the Multi-Core Era: Flexibility towards Degrees of Thread-Level Parallelism

Stijn Eyerman Lieven Eeckhout
Ghent University, Belgium
Stijn.Eyerman@elis.UGent.be, Lieven.Eeckhout@elis.UGent.be

## Abstract

The number of active threads in a multi-core processor varies over time and is often much smaller than the number of supported hardware threads. This requires multi-core chip designs to balance core count and per-core performance. Low active thread counts benefit from a few big, high-performance cores, while high active thread counts benefit more from a sea of small, energy-efficient cores.

This paper comprehensively studies the trade-offs in multi-core design given dynamically varying active thread counts. We find that, under these workload conditions, a homogeneous multi-core processor, consisting of a few high-performance SMT cores, typically outperforms heterogeneous multi-cores consisting of a mix of big and small cores (without SMT), within the same power budget. We also show that a homogeneous multi-core performs almost as well as a heterogeneous multi-core that also implements SMT, as well as a dynamic multi-core, while being less complex to design and verify. Further, heterogeneous multi-cores that power-gate idle cores yield (only) slightly better energy-efficiency compared to homogeneous multi-cores.

The overall conclusion is that the benefit of SMT in the multi-core era is to provide flexibility with respect to the available thread-level parallelism. Consequently, homogeneous multi-cores with big SMT cores are competitive high-performance, energy-efficient design points for workloads with dynamically varying active thread counts.

***Categories and Subject Descriptors*** C.1.4 [*Processor Architectures*]: Parallel Architectures

***Keywords*** Chip Multi-Core Processor; SMT; Single-ISA Heterogeneous Multi-Core; Thread-Level Parallelism

*ASPLOS '14*, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541954

## 1. Introduction

The number of active threads in a processor varies over time, and is often (much) smaller than the number of available hardware thread contexts. This observation has been made across different application domains. Desktop applications exhibit a limited amount of thread-level parallelism, with typically only 2 to 3 active threads [4]. Datacenter servers are often underutilized and seldomly operate near their maximum utilization; they operate most of the time between 10 to 50 percent of their maximum utilization level [2]. Even parallel, multi-threaded applications do not utilize all cores all the time. Threads may be waiting because of synchronization primitives (locks, barriers, etc.) and may yield the processor to avoid active spinning [6]. Finally, in a multiprogrammed environment, jobs come and go, and hence, the amount of available thread-level parallelism varies over time.

Workloads with dynamically varying active thread counts imply that multi-core chip designs should balance core count and per-core performance. Few high-performance cores are beneficial at low active thread counts, while a sea of energy-efficient cores are preferred at high active thread counts. The key question is what processor architecture is best able to deal with dynamically varying degrees of thread-level parallelism. A heterogeneous single-ISA multi-core with a few big cores and many small cores [19], might schedule threads onto the big cores in case there are few active threads, and only schedule threads on the small cores when the number of active threads exceeds the number of big cores. A conventional homogeneous multi-core with Simultaneous Multi-Threading (SMT) cores [31] might schedule threads across the various cores if there are fewer active threads than cores. Each thread would then have an entire core to its disposal, and only when the number of active threads exceeds total core count, could one engage SMT to improve chip throughput. Ideally, core count and size should be dynamically changed depending on the number of active threads, and people have proposed to fuse small cores to bigger cores as a function of the number of active threads [11, 17]. Determining the appropriate processor architecture is not only important in the context of delivering high performance under various workload conditions, it also involves other design concerns such as power/energy as well as cost to design and verify the design, i.e., a heterogeneous and core fusion processor architecture is likely more costly to design and verify than a homogeneous multi-core.

This paper studies major multi-core design trade-offs in the face of dynamically varying degrees of available thread-level parallelism. Through a set of comprehensive experiments, we find that a homogeneous multi-core with big SMT cores outperforms heterogeneous designs, under the same power envelope, when there is a varying degree of thread-level parallelism, for both multi-program and multi-threaded workloads. The intuition is that when there are few active threads, they can be scheduled across the available big cores with a few, or even a single, SMT hardware thread contexts active, and hence achieve good single-thread performance. We also find that a homogeneous multi-core with SMT performs almost as well as a heterogeneous design that also exploits SMT, and that its performance is also close to that of a dynamic multi-core design, in which the configuration (number of big and small cores) can change dynamically depending on the number of active threads.

The result that the performance of a homogeneous multi-core with big SMT cores is comparable to a heterogeneous multi-core design, we believe, is counter-intuitive. It is well-known, and confirmed by our experimental results, that a number of small cores achieve better aggregate performance (throughput) than a high-performance SMT core under the same power budget. Hence, it is to be expected that overall performance will be higher for a homogeneous multi-core with many small cores as well as for a heterogeneous multi-core with a few big cores and many small cores, when there are many active threads in the system. However, under variable active thread workload conditions, a homogeneous design with big SMT cores is a competitive design point because it can more easily adapt to software diversity, and deliver both best possible chip throughput when there are few active threads, and comparable performance when there are many active threads.

While we show that a homogeneous multi-core consisting of all big cores with SMT is competitive to a heterogeneous multi-core in terms of performance, the latter has more opportunities to save power by power-gating idle cores. Cores can only be switched off when there are fewer active threads than cores, resulting in fewer power-gating opportunities for configurations with fewer cores. We find however, that a heterogeneous multi-core has only a slightly better energy-efficiency compared to a homogeneous all-big-core configuration under variable thread-level parallelism.

The overall conclusion from this paper is that, although SMT was designed to improve single-core throughput [31], the real benefit of SMT in the multi-core era is to provide flexibility with respect to the available thread-level parallelism. Consequently, we find that a homogeneous multi-core with big SMT cores is a competitive high-performance, energy- and cost-efficient design point when the active thread count varies dynamically in the workload.

## 2. Motivation

### 2.1 Varying thread-level parallelism

We identify at least four application domains that exhibit varying degrees of available thread-level parallelism during runtime.

**Multi-programmed workloads.** The most obvious reason for having a varying degree of active threads is due to multi-programming. Jobs come and go, and hence the amount of thread-level parallelism varies over time. Jobs are also scheduled out when performing I/O (disk and network activity).

**Desktop applications.** A recent study by Blake et al. [4] quantifies the amount of thread-level parallelism in contemporary desktop applications. They find the amount of thread-level parallelism to be small, with typically only 2 to 3 active threads on average, even after ten years of multi-core processing.

**Server workloads.** Servers in datacenters operate between 10 to 50 percent of their maximum utilization level most of the time according to Barroso and Hölzle [2]. They found the distribution of utilization at a typical server within Google to have a peak around zero utilization and 30 percent utilization. A multi-core server that is underutilized implies that there are only few active threads.

**Multi-threaded applications.** Even multi-threaded applications may not have as many active threads as there are software threads at all times during the execution. Threads may be waiting because of synchronization due to locks, barriers, etc., and may yield to the operating system to avoid active spinning. Figure 1 quantifies the number of active threads when running the PARSEC benchmarks [3] on a twenty-core processor. (We refer to Section 3 for details on the experimental setup.) Some benchmarks have 20 active threads most of the time (blackscholes, canneal and raytrace), whereas others have 20 active threads only a small fraction of the time (e.g., ferret, freqmine and swaptions). Some benchmarks have either one or twenty active threads (e.g., bodytrack and swaptions), others have a larger variation in the number of active threads (e.g., dedup, ferret and freqmine). On average across all PARSEC benchmarks running on 20 cores, we find that there are 20 active threads only half of the time, and 31% of the time, only 4 or fewer threads are active. Note that these numbers are generated for the parallel part of the application — the so-called region of interest (ROI) as it is defined for the PARSEC benchmarks — so the limited number of active threads only stems from inter-thread synchronization during parallel execution, and is not due to other sequential code such as initialization.

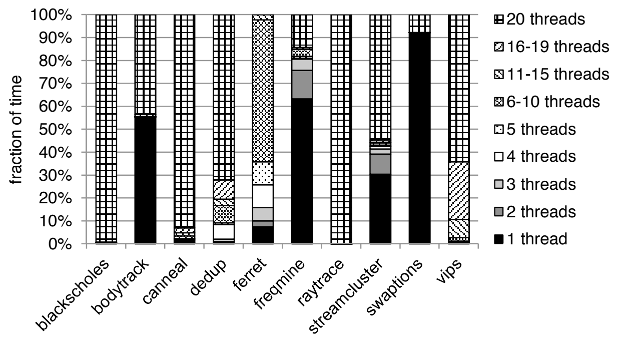


**Figure 1.** Distribution of the number of active threads for the PARSEC benchmarks on a twenty-core processor.

### 2.2 Multi-core design choices

There exist three major multi-core architectures: symmetric or homogeneous, asymmetric or heterogeneous, and dynamic [10]. All cores in a homogeneous multi-core have the same organization; examples are the Intel Sandy Bridge CPU [25], AMD Opteron [15], IBM POWER7 [14], etc. Each core typically implements Simultaneous Multi-Threading (SMT), effectively providing a many-thread architecture, e.g., an 8-core processor with 4 SMT threads per core effectively yields a 32-threaded processor.

A heterogeneous (or asymmetric) multi-core features one or more cores that are more powerful than others. In case of a single-ISA heterogeneous multi-core, there are so-called big, high-performance cores and small, energy-efficient cores. NVidia's Kal-El [22] integrates four performance-tuned cores along with one energy-tuned core, and ARM's big.LITTLE [8] combines a high-performance core with a low-energy core.

A dynamic multi-core is able to combine a number of cores to boost performance of sequential code sections. Core fusion [11, 17] dynamically morphs cores to form a bigger, more powerful core. Thread-level speculation and helper threads [9, 28], in which assist-threads running on other cores help speeding up another thread, could also be viewed as a form of dynamic multi-core. Recently, Khubaib et al. [16] propose MorphCore, which is a high-performance out-of-order core that can morph into a many-threaded in-order core when the demand for parallelism is high.

### 2.3 Goal of this paper

Given the background in workloads and the multi-core design space as just described, the following key question arises: ***How to best design a single-ISA multi-core processor in light of varying degrees of thread-level parallelism in contemporary workloads?*** As mentioned in the introduction, all three design options can deal with varying numbers of active threads, one way or the other. A homogeneous multi-core can distribute the active threads across the various cores and only activate SMT when there are more active threads than cores. A heterogeneous multi-core can schedule the active threads on the big cores and only schedule threads on the small cores when there are more active threads than big cores. A dynamic multi-core can form as many cores as there are active threads. However, without a detailed and comprehensive study, it is unclear which multi-core architecture paradigm yields best performance under varying active thread counts. This paper, to the best of our knowledge, is the first to explore this multi-core design space and comprehensively compare multi-core paradigms in light of variable active thread count. Note that specialized accelerators are not in this paper's scope, as we focus on single-ISA multi-cores.

| | Big core | Medium core | Small core |
|---|---|---|---|
| Frequency | 2.66GHz | 2.66GHz | 2.66GHz |
| Type | Out-of-Order | Out-of-Order | In-Order |
| Width | 4 | 2 | 2 |
| ROB size | 128 | 32 | N/A |
| Func. units | 3 int, 2 ld/st | 2 int, 1 ld/st | 2 int, 1 ld/st |
| | 1 mul/div | 1 mul/div | 1 mul/div |
| | 1 FP | 1 FP | 1 FP |
| SMT contexts | up to 6 | up to 3 | up to 2 |
| L1 I-cache | 32KB | 16KB | 6KB |
| | 4-way assoc | 2-way assoc | 2-way assoc |
| L1 D-cache | 32KB | 16KB | 6KB |
| | 4-way assoc | 2-way assoc | 2-way assoc |
| L2 cache | 256KB | 128KB | 48KB |
| | 8-way assoc | 4-way assoc | 4-way assoc |
| Last-level cache | 8MB, 16-way assoc | | |
| On-chip interconn. | 2.66GHz, full cross-bar | | |
| DRAM | 8 banks, 45ns access time | | |
| Off-chip bus | 8GB/s | | |

**Table 1.** Big, medium and small core configurations.

## 3. Experimental Setup

### 3.1 Multi-core design space

To evaluate the various multi-core paradigms in the context of varying thread counts, we use the following experimental setup. We consider three types of cores: a four-wide out-of-order core (big core), a two-wide out-of-order core (medium core), and a two-wide in-order core (small core), see also Table 1 for more details about these microarchitectures.

We compare all multi-core architectures under the (approximate) same power envelope. We therefore estimate power consumption using McPAT [20] (assuming 45 nm technology and aggressive clock gating). The big core consumes approximately 1.8 times the power of the medium two-wide OoO core on average, and 4.4 times the power of the small two-wide in-order core. We conservatively assume that one big core is power-equivalent to two medium cores and five small cores. We validate later in this section that these scaling factors result in an approximately equal power consumption, even when the big cores execute six threads through SMT (which leads to higher utilization and therefore higher dynamic power consumption). When evaluating energy efficiency in Section 7, we assume idle cores are power gated.

We keep total on-chip cache capacity constant when exploring the multi-core design space, in order to focus on the impact of core types and organization, and not cache capacity. This implies that we have to set the private cache size of the medium core two times smaller compared to the big core, and five times for the small core, see also Table 1. (We pick numbers that are powers of two or just in between two powers of two). The last-level cache (LLC) is shared across all cores, and has the same size for all multi-core configurations (8MB). The on-chip network is a full crossbar between all cores and the shared LLC. Although not realistic, a full crossbar ensures that the results are not skewed in favor of the few large cores configuration, which would experience less contention in the on-chip network compared to a many small cores configuration. We use the multi-core simulator Sniper [5] enhanced with cycle-level out-of-order and in-order core models, as well as SMT support.

Total chip power budget is equivalent to 4 big cores or 8 medium cores or 20 small cores, plus a shared LLC. This allows for 9 possible designs, see Figure 2. (For the heterogeneous designs, we only consider mixes of big cores and medium cores or small cores; we do not consider mixes of medium and small cores). In the remainder of the paper, these designs are referred to as 4B, 3B2m, 3B5s, 2B4m, 2B10s, 1B6m, 1B15s, 8m and 20s, as indicated in the figure. 4B, 8m and 20s are homogeneous multi-cores (all cores of the same type), while the others are heterogeneous. With SMT enabled, we assume that a big core is able to execute up to six threads; a medium core can execute up to three threads; and a small in-order core can execute up to two threads (using fine-grained multithreading), so that all configurations can run up to 24 threads. The SMT core that we simulate implements static ROB partitioning and a round-robin fetch policy [24].

The average total (static plus dynamic) power consumption of the three homogeneous configurations running 24 threads is 46 Watt for 4B, 50 Watt for 8m, and 45 Watt for 20s (averaged across all homogeneous multi-program workloads, see later). The power consumption of the heterogeneous configurations varies between 46 and 50 Watt. This justifies our claim that all configurations operate more or less under the same power envelope.

## 3.2 Workloads

***Multi-program workloads.*** We consider multi-program workloads using the SPEC CPU 2006 benchmarks with their reference inputs. In order to limit the number of simulations, we select 12 representative benchmark-input combinations. The selection is based on the relative performance of the benchmarks on the three core types. We evaluated all 55 SPEC CPU 2006 benchmark-input combinations on the three core designs (big, medium and small) and calculated relative performance with respect to the big core. We then picked 12 benchmarks that cover the full performance range, i.e., the benchmarks that have the highest and lowest relative performance, along with in-between benchmarks picked such as to provide good coverage.

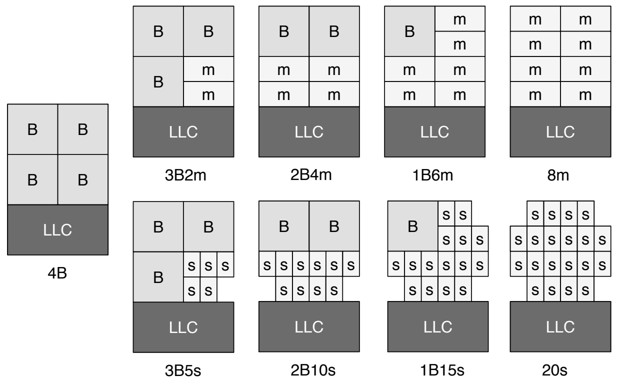


**Figure 2.** The nine power-equivalent multi-core designs considered in this study (B=big core, m=medium core, s=small core).

For each benchmark, we take a 750 million instruction single simulation point to reduce simulation time [26]. When running a multi-program workload, we stop the simulation when all of the programs have executed at least 750 million instructions, thereby restarting programs that reached the end of the 750 million instruction simulation point. We summarize multi-program performance using the system throughput (STP) metric [7] or weighted speedup [27], which is a measure for the number of jobs completed per unit of time. For computing STP, we normalize against isolated execution on the big core. When reporting STP numbers averaged across a set of workloads, we use the harmonic mean because STP is a rate metric (inversely proportional to time). We also calculate average normalize turnaround time (ANTT [7]) to show the impact of the multi-core design on per-program performance.

We evaluate homogeneous multi-program workloads (multiple copies of the same benchmark) as well as heterogeneous multi-program workloads (different benchmarks co-run). We vary the number of programs from 1 to 24. For the heterogeneous multi-program workloads, we randomly construct 12 two-, 12 three-, 12 four-, etc., up to 12 twenty-four-thread combinations, while making sure that every benchmark is included an equal number of times for all thread counts. Velasquez et al. [32] show that this balanced random sampling technique is more representative compared to fully random sampling.

We intentionally limit the number of active threads to 24 to reflect a (realistic) situation with a modest and variable thread count. Given the hardware budget of 4 big cores, this is already a considerable number of threads (6 threads per core). Our results confirm that at a (constantly) large thread count, a design with many small cores is optimal, but in this study we specifically target those workloads that exhibit a variable active thread count. Furthermore, we believe our results are general enough to be projected to larger hardware budgets and thread counts (e.g., 8 large cores and up to 48 threads).

Scheduling also plays an important role in multi-program workload performance. A general principle that we maintain is to first schedule threads on the big core(s) in a heterogeneous design before scheduling on the small cores. Likewise for SMT, we first distribute threads across cores before engaging SMT, e.g., when there are fewer active threads than cores, we run each thread on a separate core, but when there are more active threads than cores, we need to co-run threads on a single core through SMT. A heterogeneous design also implies deciding which thread to execute on which core. Similarly, in the case of SMT, we need to decide which threads to co-run on a core, since different co-runner schedules may have significant impact on performance [27]. As exploring all possible combinations of program schedules is infeasible because of simulation time considerations, we use offline analysis for determining the best possible schedule. We run each benchmark on each of the different core types in isolation, and use this analysis to steer application-to-core mapping for the heterogeneous design points for best performance. Likewise for SMT, we run all possible two-, three-, etc., up to six-program combinations on the big core (up to four for the medium and two for the small cores), and select the best possible co-schedule. This approach ignores the impact of resource sharing among cores to steer scheduling, however, we do account for resource sharing (shared cache, memory bandwidth, etc.) during detailed simulation for the selected schedules.

***Multi-threaded workloads.*** We also evaluate multi-threaded workloads, using the PARSEC benchmarks [3]. We vary the number of threads from 4 to 24 in steps of 4. We only included the benchmarks that allow for a number of threads that is not fixed to a power of 2. We use the medium size input set for all benchmarks, and evaluate the execution time for the parallel part only (the so-called region of interest or ROI) and for the whole program (including the sequential initialization and finalization code). We report speedups versus a four-threaded execution on the 4B configuration.

## 4. Multi-Program Workloads

We now evaluate the performance of the nine multi-core designs for multi-program workloads, i.e., workloads consisting of multiple single-threaded programs. (We will discuss multi-threaded workloads in the next section.) We first discuss performance as a function of thread count, and subsequently compute aggregate performance under the assumption of various active thread count distributions.

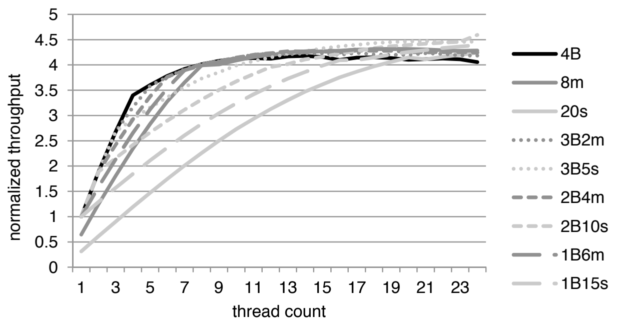
### 4.1 Performance as function of thread count

Figure 3 shows average performance for the nine multi-core configurations as a function of the number of threads from 1 to 24. All designs have SMT enabled in all cores, the non-SMT curves can be reconstructed by leveling off performance as soon as thread count equals core count. The interesting observation is that the homogeneous 4B configuration performs well compared to the other homogeneous and heterogeneous designs. Although the heterogeneous designs outperform the 4B configuration for some thread counts, 4B performs well over the full range of thread counts. When thread count is low, 4B yields the highest performance, and when thread count is high, 4B yields only slightly lower performance compared to the many medium and small core designs (8m and 20s).

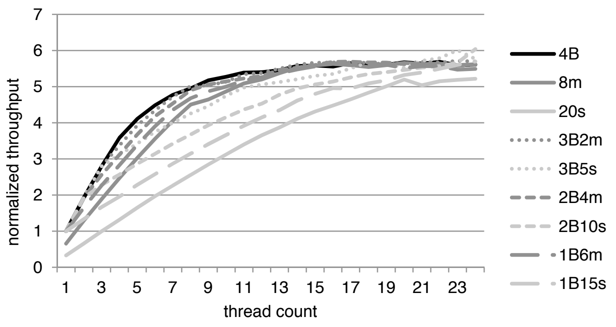
It is not surprising that multi-core configuration 4B performs well for low thread counts: for 4 or fewer threads, each powerful big core has only one thread running. What is more remarkable is that the SMT multi-core performs also relatively well when thread count is high: for example, when there are 24 threads, each core executes six threads concurrently, but performance is close to that of running 24 threads on 8 medium cores (each three-way SMT) or 24 threads on 20 small cores (4 cores use 2-way SMT, the others execute only one thread).

To explain this behavior, Figure 4 shows the same graphs for two homogeneous workloads, which were picked to illustrate the interesting diversity observed across the various benchmarks; we found the benchmarks to roughly classify into these two categories. Tonto (left graph) shows the intuitively expected behavior: up to 8 threads, performance of the SMT multi-core is better than or similar to the performance of the heterogeneous architectures, but beyond 8 threads, its performance is inferior. Tonto clearly benefits from the higher aggregate execution resources available in the heterogeneous design points as well as in the homogeneous multi-core with all medium or small cores at high active thread counts. For libquantum (right graph) on the other hand, the multi-core with SMT performs approximately as well as the other design points for high thread counts. What happens here is that as the number of threads increases, more and more pressure is put onto the shared resources (shared last-level cache, memory bandwidth, DRAM banks, etc.), upto the point that performance gets largely dominated by shared resource contention and less by individual core performance. In particular, we observe that, for libquantum, memory access time is 4 times higher for 24 threads than for one isolated thread for both the 20s and 4B configurations due to contention on the memory bus. This tightens the gap and flattens out the performance differences between the various multi-core configurations.

It is interesting to note that, at high thread counts, the performance of 4B versus the other design points is slightly smaller for the homogeneous workloads than for the heterogeneous workloads, compare graphs (a) versus (b) in Figure 3. In fact, for heterogeneous workloads and 24 threads, we notice that the performance of 4B is only 7.1% lower than the maximum (2B10s), while for homogeneous workloads, 4B's performance is 11.6% lower than

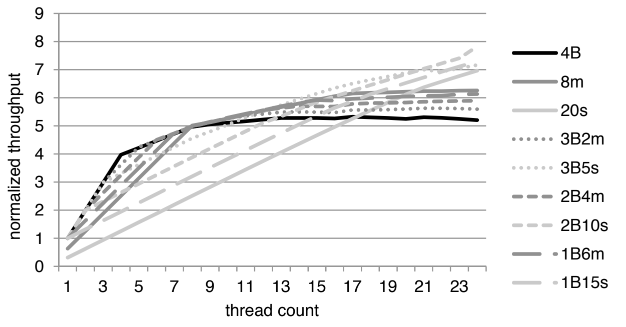


(a) Homogeneous workloads

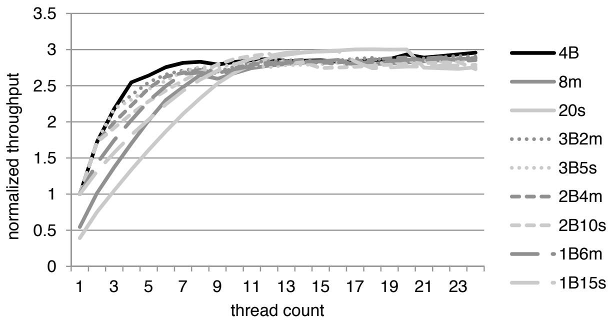


(b) Heterogeneous workloads

**Figure 3.** Comparing the performance for the nine multi-core design points with homogeneous and heterogeneous multi-program workloads.



(a) tonto



(b) libquantum

**Figure 4.** Performance of the nine multi-core design points for two representative benchmarks (both homogeneous multi-program workloads): (a) tonto and (b) libquantum.

the maximum (2B10s). This is due to the fact that heterogeneous workloads consist of mixes of both memory-intensive and compute-intensive benchmarks. Scheduling a memory-intensive benchmark with compute-intensive benchmarks on one core using SMT enables the memory-intensive benchmark to occupy a larger fraction of the core's private cache (256KB in our study), as the compute-intensive benchmarks are less demanding for cache space. In case of a multi-core with many small cores (20s), each core has a small private cache (48KB in our setup), hence, a memory-intensive benchmark would not get as much cache space. By intelligently scheduling benchmarks to cores and SMT thread contexts, the 4B multi-core is better capable of utilizing cache space than a multi-core with many small cores and relatively smaller private caches.

For completeness, Figure 5 shows the average normalized turnaround time (ANTT) for the homogeneous workloads as a function of thread count (the results for heterogeneous workloads are similar). At small thread counts, the 4B design results in the lowest per-program execution time (highest per-program performance), because all threads can run on a big core. Per-program execution time increases as thread count goes up, because more threads share a core through SMT, reducing per-program performance. For the other extreme configuration 20s, the turnaround time is larger for low thread counts, because of the poorly performing cores, but it remains more stable as thread count increases, due to a smaller degree of sharing. The conclusions are similar to that of the througput results: at low thread counts, 4B has the highest throughput and the lowest per-program execution time, and at high thread counts, the configurations with more and smaller cores have the highest throughput and the lowest per-program execution time, but the 4B configuration remains close.

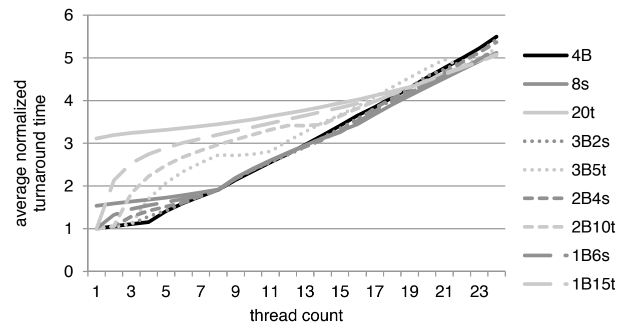


**Figure 5.** Comparing the ANTT for the nine multi-core design points with homogeneous multi-program workloads.

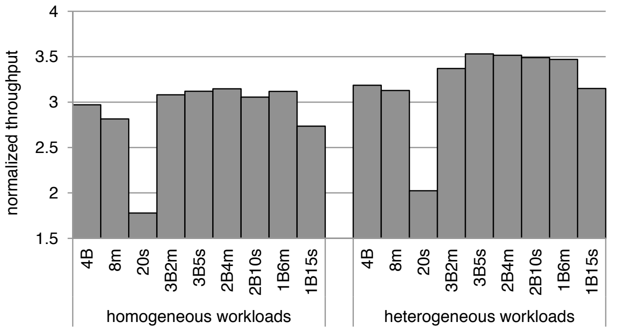


**Figure 6.** Average performance assuming a uniform thread count distribution and no SMT.

We conclude this section with our first finding:

**Finding #1:** *A homogeneous multi-core consisting of all big SMT cores yields better performance than a heterogeneous multi-core for a small number of threads* (due to the bigger cores) *and only slightly worse for a large number of threads* (because shared resource contention largely dominates performance for workload mixes of memory-intensive applications, and cache capacity can be used more efficiently through intelligent scheduling).

### 4.2 Thread count distributions

We now compare the multi-core designs under various active thread distributions, assuming uniform distributions as well as distributions observed in datacenter operations.

#### 4.2.1 Uniform distribution.

We begin with assuming a uniform distribution over 24 threads, i.e., each thread count (1 to 24 threads) has equal probability.

***No SMT.*** We first assume that none of the cores implement SMT. Figure 6 shows the average performance for all of the multi-core designs *without* SMT. Each core can execute only one thread at a time, and when there are more threads than cores, multiple threads run on one core sequentially through time-sharing.

Clearly, the 4B configuration outperforms the other *homogeneous* configurations (8m, 20s). Being able to execute faster at low thread counts is more important than achieving a high throughput at high thread counts. This is in line with Amdahl's law: as parallelism increases, the performance of the sequential part (low thread count) dominates the performance of the program as a whole.

The most important conclusion is that the optimal design without SMT is 2B4m for homogeneous workloads and 3B5s for heterogeneous workloads — both heterogeneous multi-core designs. Hence our second finding:

**Finding #2:** *In the absence of SMT, heterogeneous multi-cores outperform homogeneous multi-cores across varying thread counts.* At low thread counts, the big cores in a heterogeneous multi-core can be used to get high performance, while at high thread counts, the larger amount of small cores can be used to exploit thread-level parallelism. This is in line with recent work that advocates single-ISA heterogeneous multi-core processors [10, 19].

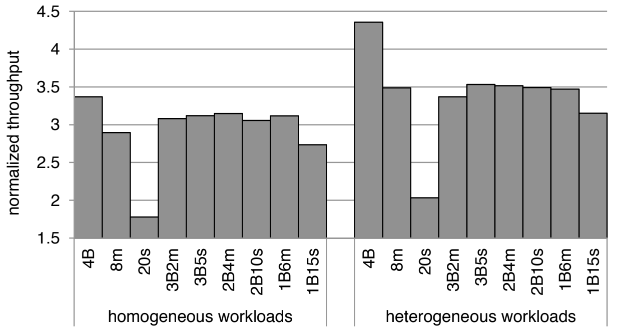


**Figure 7.** Average performance assuming a uniform thread count distribution and SMT in the homogeneous configurations.

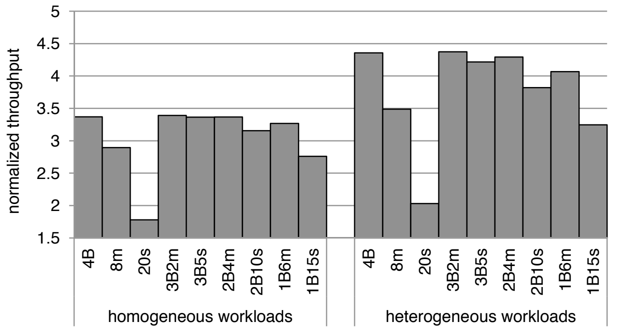


**Figure 8.** Average performance assuming a uniform thread count distribution and SMT in all configurations.

***SMT in homogeneous designs.*** We now assume SMT is implemented in the homogeneous designs (4B, 8m and 20s), but not the heterogeneous designs. Figure 7 shows average performance for the various designs. It is interesting to compare this graph against the one in Figure 6, which showed that heterogeneous multi-cores yield higher performance than homogeneous multi-cores when the number of threads varies. Now, through Figure 7, we observe that by adding SMT to the homogeneous multi-cores, the 4B design outperforms the other designs. This leads to:

**Finding #3:** *A homogeneous multi-core with big SMT cores outperforms a heterogeneous multi-core (without SMT) under the same power budget. Put differently, SMT outperforms heterogeneity as a means to cope with varying thread counts.* The intuition is that, at low thread counts, the 4B design with SMT is able to use all 4 big cores, while the number of big cores in the heterogeneous designs is always smaller. At high thread counts, a homogeneous multi-core with big SMT cores allows for more concurrent threads (24 in total) compared to heterogeneous multi-cores (at most 20 in the

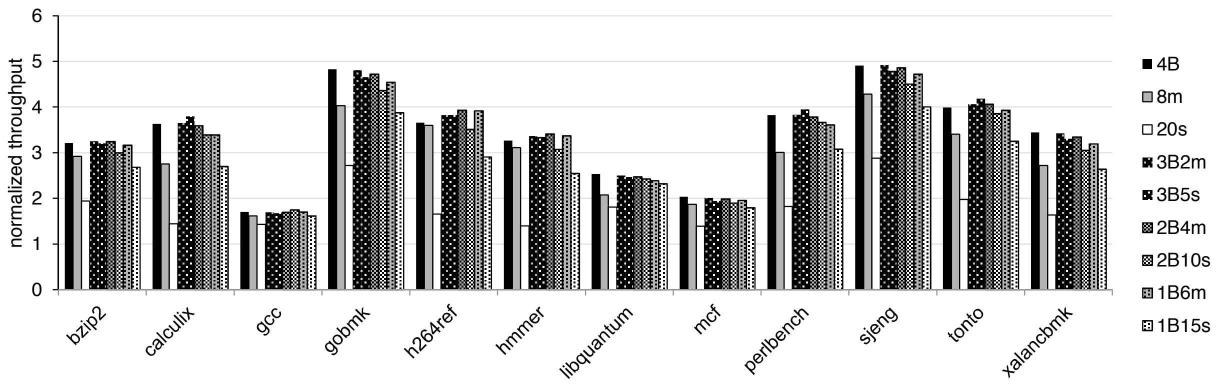
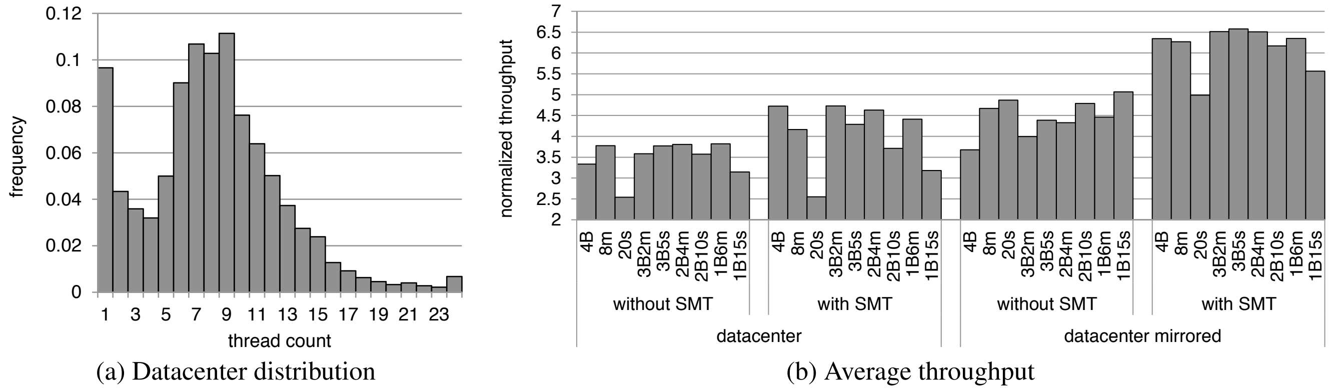


**Figure 9.** Average performance per benchmark assuming a uniform thread count distribution.



(a) Datacenter distribution

(b) Average throughput

**Figure 10.** Datacenter distribution and average performance using the datacenter distribution and the mirrored datacenter distribution.

20s design point), yielding higher overall throughput within the same power budget.

***SMT in all designs.*** Finally, Figure 8 shows average performance when SMT is enabled in all cores of all designs. For homogeneous workloads, the performance for the best heterogeneous configuration is 5.6% higher than that of 4B *without* SMT in all configurations (Figure 6), but only 0.6% higher than *with* SMT in all designs (Figure 8). For heterogeneous workloads, the homogeneous 4B design even outperforms the best heterogeneous design by 0.5%. Thus, in other words:

**Finding #4:** *The added benefit of combining heterogeneity and SMT is limited.*

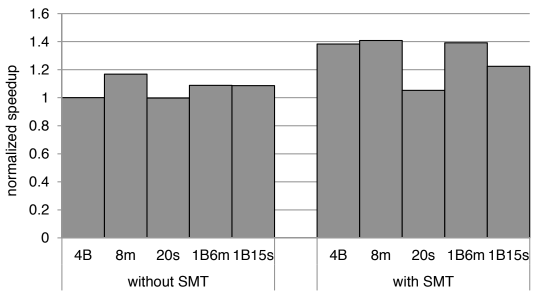
It is also interesting to observe that the optimal heterogeneous design shifts from 2B4m without SMT to 3B2m with SMT for the homogeneous workloads, and from 3B5s to 3B2m for the heterogeneous workloads. Hence:

**Finding #5:** *Adding SMT to the heterogeneous designs makes the optimum shift towards fewer and larger cores.* This is in line with the general observation that SMT in larger cores enables flexibility as a function of active thread count.

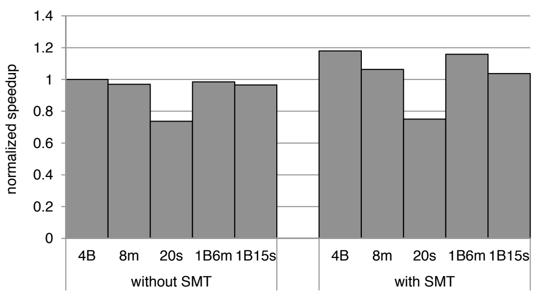
***Per-benchmark results.*** Figure 9 shows average performance for the various multi-core configurations (SMT enabled in all cores) for each benchmark, assuming a uniform distribution. The results vary across benchmarks: for some benchmarks (calculix, h264ref, hmmer and tonto), 4B performs worse than the best heterogeneous multi-core, while for others it performs similarly, or even slightly better (libquantum and mcf). Detailed analysis of the results revealed that the latter category of benchmarks have high memory bandwidth demands, resulting in bandwidth-bound performance numbers for high thread counts. Section 8.2 contains results with a higher memory bandwidth setting.

### 4.2.2 Datacenter distributions.

Figure 10(b) shows average performance across two different thread count distributions, assuming heterogeneous workload mixes. "Datacenter" is the distribution taken from [2] for CPU utilization in a datacenter, adapted to a workload of at most 24 threads; Figure 10(a) shows the distribution: there is a peak at 1 thread (low utilization) and one at 7 to 9 threads (30%-40% utilization). "Mirrored datacenter" is the same distribution, mirrored around the center. This means that there now is a peak at 24 threads, and one around 16 to 18 threads. We use this distribution to model a more heavily



(a) ROI only



(b) Whole program

**Figure 11.** Average normalized speedup for all PARSEC benchmarks.

loaded server park, with a distribution skewed to the higher thread counts.

For the datacenter distribution, 1B6m is the best performing configuration without SMT, see Figure 10(b). This is as expected: we have 1 big core for the peak at 1 thread, and 7 cores in total to cover the peak around 7 threads. Adding SMT again makes the fewer but bigger cores configurations more optimal, with the best performance for the 4B configuration. For the mirrored datacenter distribution, the optimum without SMT is 1B15s, because there is a peak at 16 threads. For the configurations with SMT, 3B2m is optimal, with 4B performing only 0.6% worse.

**Finding #6:** *For distributions that are skewed to fewer threads, the 4B configuration with SMT is optimal. For distributions that are skewed towards more active threads, 4B with SMT becomes less optimal, but its performance is very close to the optimum.*

## 5. Multi-Threaded Workloads

As discussed in Section 2, multi-threaded programs can also have a variable number of active threads. When threads have to wait due to synchronization (e.g., a barrier), they can be scheduled out by the operating system to free resources for other runnable threads. Periods with low active thread count are critical to performance, since they exhibit little parallelism and are therefore more difficult to speed up [6]. Achieving high performance at low thread counts is therefore likely to be even more crucial for multi-threaded workloads than for multi-programmed workloads.

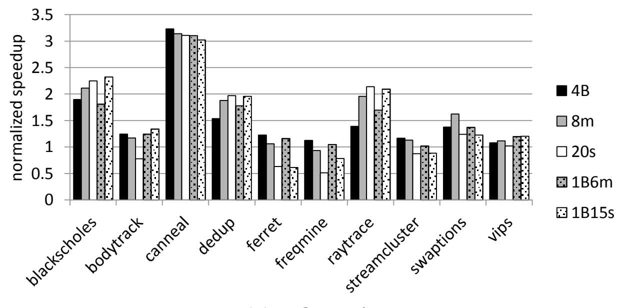
We use the PARSEC benchmarks in this section, and always report the maximum speedup across all possible thread counts. Note this does not necessarily equal total core count because of interference between threads in shared resources. We further assume pinned scheduling which pins threads to cores to improve data locality (as done in modern multi-core schedulers [13]); and we execute serial phases on the big core when reporting whole program performance results. We limit the discussion in this section to heterogeneous designs with a single big core as we assume pinned scheduling which does not enable benefiting from multiple big cores. (We verified that none of the other heterogeneous designs have larger speedups than the ones reported here.)

The results, averaged across all benchmarks, are shown in Figure 11. We split up the results for the ROI-only and the whole program, and show the speedups without SMT (i.e., number of threads equal to number of cores) and with SMT. For the ROI-only results without SMT, 8m is the optimal design. This is because most of the applications scale well up to 8 threads, but not beyond. Adding SMT boosts the speedup for the 4B design, and makes its speedup very close to that of 8m. Overall, the 4B design with SMT performs well for benchmarks that have poor parallelism, and performs only slightly worse for programs that scale well.

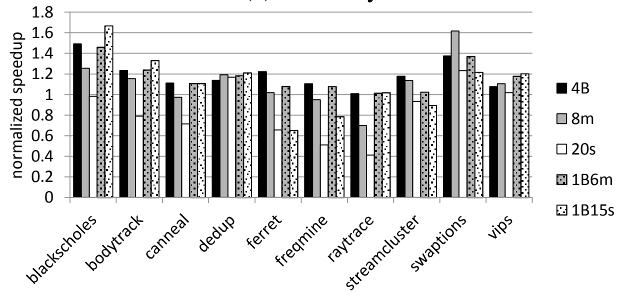
For the whole-program results, the 4B design performs best both without and with SMT. The 4B design performs best for applications with limited parallelism, and close to optimal for applications that scale better, but that have a large initialization and finalization serial phase. Without SMT, the heterogeneous designs perform close to the 4B configuration: they speed up the serial phases, but on average, the poorly scaling benchmarks achieve better performance on the 4B configuration and this dominates the average. With SMT enabled, the difference between the 4B configuration and the heterogeneous configurations is larger, because 4B with SMT speeds up well-scaling benchmarks more.

Figure 12 shows per-benchmark speedups. For the ROI-only results (top graph), it clearly shows the difference across benchmarks: 20s is optimal for well-scaling benchmarks, while 4B or a heterogeneous design are optimal for poorly scaling benchmarks. For the whole program results (bottom), the optimal configuration is 4B or a heterogeneous design for most of the benchmarks.

**Finding #7:** *SMT is also beneficial for multi-threaded workloads. As for the multi-program workloads, adding SMT lets the optimal design shift to fewer but larger cores. A homogeneous design with big SMT cores outperforms the best heterogeneous design without SMT, and performs close to, and sometimes even slightly better than, the best heterogeneous design with SMT.*

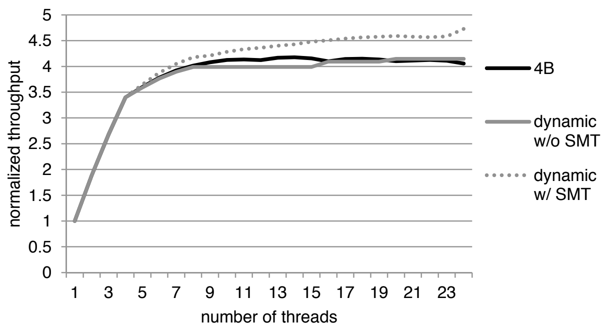


(a) ROI-only

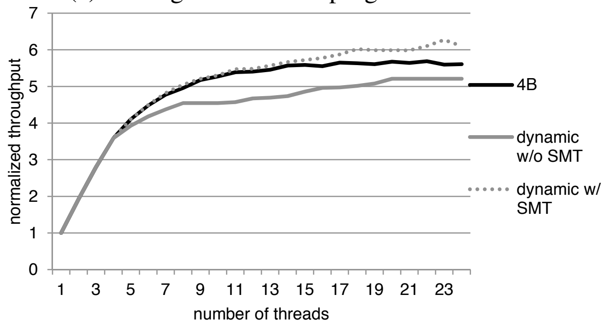


(b) Whole program

**Figure 12.** Normalized speedup for the individual PARSEC benchmarks.



(a) Homogeneous multi-program workloads



(b) Heterogeneous multi-program workloads

**Figure 13.** Throughput as a function of the number of threads for the 4B configuration with SMT and the dynamic core fusion configuration with and without SMT.

## 6. Dynamic Multi-Cores

Dynamic multi-cores are multi-core processors with a dynamic configuration [11, 17]: core configuration and the number of cores can dynamically vary between many small cores and a few large cores, and in-between heterogeneous configurations. Theoretical studies, such as the one of Hill and Marty [10], show that this type of multi-core is optimal in the context of varying parallelism and varying thread count. Through dynamic adaptation, one or a few big cores can be formed when there is low parallelism, while the configuration is changed to many small cores when there are a lot of active threads. This technique is essentially the inverse of SMT: an SMT core executes a single thread but can execute multiple threads at higher active thread counts; a dynamic multi-core executes threads on independent cores, which can be fused to bigger cores at low active thread counts.

To compare the abilities of a homogeneous multi-core with big SMT cores (4B) versus a dynamic multi-core to cope with varying active thread counts, we assume an ideal dynamic multi-core that can be morphed without overhead into any of the 9 multi-core configurations in Figure 2. This ideal dynamic multi-core chooses the best performing configuration (out of the 9 possible configurations) at each thread count for each workload. This is an optimistic assumption in favor of dynamic multi-cores, since fusing cores is likely to involve a non-negligible time, area and power overhead. Figure 13 compares dynamic multi-cores (both with and without SMT) against the 4B configuration (with SMT) for the homogeneous and heterogeneous multi-program workloads. This figure shows that dynamic multi-cores without SMT yield similar or even worse overall performance. Especially for heterogeneous workloads, SMT seems to perform better than a dynamic multi-core design. The reason is that SMT enables better utilization and higher throughput within the same power budget, especially when the programs are complementary in their resource demands. SMT also allows for more fine-grained parallelism: for the dynamic multi-core, a big core can be split up into 2 medium cores or 5 small cores, but an SMT core can also execute 3 and 4 threads concurrently, while fully utilizing all resources. As a result, the 4B line in Figure 13(b) smoothly increases, while the dynamic line (without SMT) shows multiple plateaus with jumps when the configuration changes. A dynamic multi-core that also supports SMT performs the best, but this will probably result in a very complex design and an even more complex scheduling and reconfiguration policy. We thus conclude:

**Finding #8:** *Homogeneous multi-cores with big SMT cores outperform (or are at least competitive to) dynamic multi-cores as a way to cope with variable active thread counts. A combination of both is optimal, but is also the most complex, both with respect to design and run-time scheduling.*

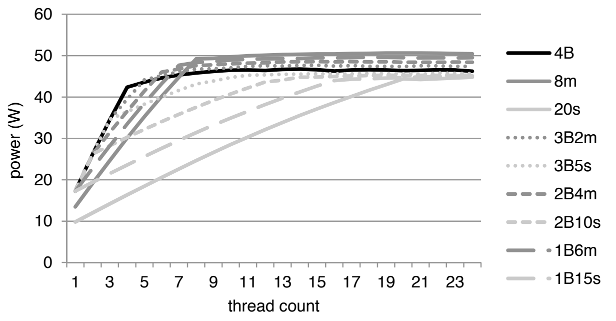


**Figure 14.** Power consumption as a function of thread count for all configurations assuming power gating.

## 7. Energy Efficiency

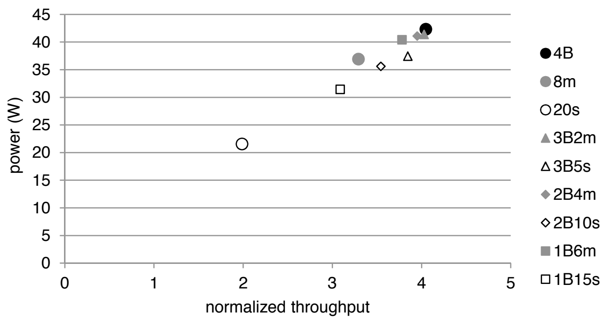
In the previous sections, we focused on performance under an equal total power budget. However, power-gating can be used to turn off idle cores, resulting in lower power consumption at low active thread counts. Especially for the configurations with many medium or small cores, this may result in improved power/energy-efficiency compared to the homogeneous configuration with a few big SMT cores.

***Power consumption as a function of thread count.*** Figure 14 shows average power consumption for all configurations (all configurations have SMT enabled in all cores) as a function of thread count when power-gating unused cores (averaged across all homogeneous multi-program workloads). It is interesting to study power consumption along with performance as shown in Figure 3: the 4B configuration consumes most power at low active thread counts while delivering highest performance; the 20s configuration consumes least power while delivering poorest performance; on the other hand, at high thread counts, all configurations perform nearly as well while consuming similar levels of power.

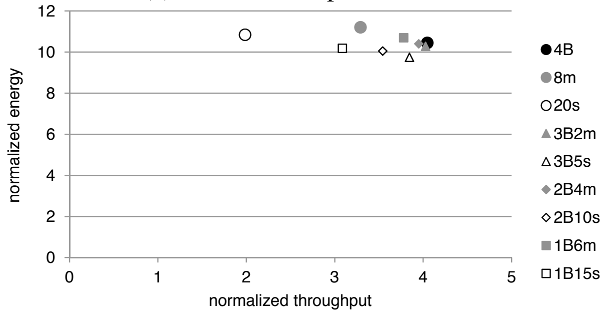
Figure 14 also shows that activating SMT contexts increases power consumption, due to the increase in resource utilization, but not as much as the increase in power consumption from activating cores (see for example the 4B configuration: power consumption increases from 42 Watt for 4 threads to 46 Watt for 24 threads). Note that the numbers for one thread (leftmost points) do not show the 1/2/5 relative power difference for the big, medium and small cores (the power consumption for one active core is 17.3, 13.5 and 9.8 Watt, for B, m and s, respectively). This is because the shared L3 cache and the main memory (DRAM) are active all the time, irrespective of active thread count — these resources consume approximately 7 Watt. The relative difference in power consumption for the three core types is reflected in the slopes of the 4B, 8m and 20s configurations (part of the curves that do not use SMT, i.e., with thread count lower than or equal to core count).

***Pareto-optimal designs.*** Figure 15 shows the power and energy consumption as a function of performance for the heterogeneous multi-program workloads (assuming a uniform thread count distribution). There are several interesting observations to be made. First, the 20s configuration consumes the least power, but results in high energy consumption due to its poor performance. In other words, a configuration with many small cores is not energy-optimal. Second, the 4B configuration is the best performing, but also has higher power and energy consumption. Third, the Pareto-optimal frontier is populated with heterogeneous design points, along with the best-performance 4B and lowest-power 20s configurations: the Pareto-optimal frontier consists of the following design points, 4B, 3B2m, 2B4m, 3B5s, 2B10s, 1B15s and 20s, for power versus performance (top graph in Figure 15), and 4B, 3B2m and 3B5s, for energy versus performance (bottom graph). In other words, heterogeneity trades off performance for power and energy consumption. The design point with the minimum energy-delay product (EDP) across all the designs considered is the 3B5s configuration, yet this heterogeneous design point improves EDP by as little as 4.1% and 1.8% over the 4B design point for the homogeneous and heterogeneous workloads, respectively. This leads to the following finding:

**Finding #9:** *Heterogeneous multi-core designs, when power gating idle cores, yield an (only) slightly better energy-efficiency compared to homogeneous multi-cores with big SMT cores under variable active thread count conditions.*



(a) Power versus performance



(b) Energy versus performance

**Figure 15.** Throughput versus power (top) and energy (bottom) consumption for heterogeneous multi-program workloads (assuming a uniform thread count distribution).

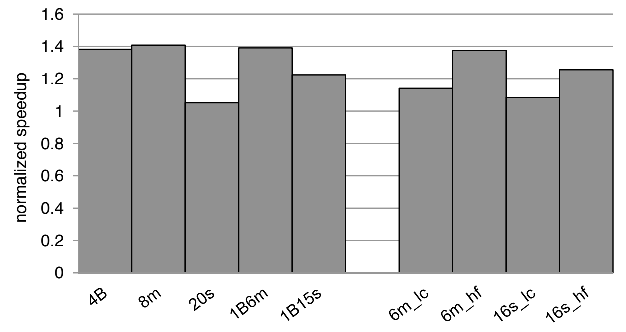


**Figure 16.** Average multi-threaded benchmark performance with alternative large-cache and high-frequency configurations.

## 8. Alternative Multi-Core Designs

### 8.1 Larger caches or higher frequency for the small cores

In Section 3, we assumed particular design decisions that may impact the final results. One decision was to keep total cache capacity constant across all designs. The motivation was to evaluate the impact of core type and organization, not cache capacity. Nevertheless, we noticed that sharing a cache between multiple programs co-executing on an SMT core can lead to better cache usage. We therefore now evaluate the effect of keeping private cache sizes constant across core types. We also evaluate the impact of increasing frequency of the small cores to improve its performance.

Figure 16 shows average speedup for the multi-threaded benchmarks (ROI-only). 6m_lc and 16s_lc (lc stands for larger cache) are configurations where the private L1 and L2 cache sizes for the medium and small cores are equal to that of the big core. Larger caches consume more power, leading to a different power-equivalence among core types: a big core is now power-equivalent to 1.5 medium cores and 4 small cores, which explains the decreased core count for the configurations with a larger cache. Further, the 6m_hf and 16s_hf configurations contain 6 medium cores or 16 small cores with clock frequency increased from 2.66 GHz to 3.33 GHz. This increase in frequency also results in a 1 to 1.5, and a 1 to 4 power-equivalence between the big and medium cores, and the big and small cores, respectively.

The results in Figure 16 show that a larger cache and, more distinctly, higher frequency leads to a higher speedup for the small-core configuration (compare 16s_lc and 16s_hf versus 20s). This is because many benchmarks do not scale well up to 20 threads, and reducing core count in exchange for more cache capacity or a higher frequency results in higher speedup. For the medium-core configuration (8m) on the other hand, enlarging the cache or increasing the frequency has a negative impact on performance: the benefits of a large cache or a higher frequency do not compensate for the reduction in core count. Overall, we observe that a homogeneous multi-core with big SMT cores achieves best performance for the given power budget. Hence, we conclude that:

**Finding #10:** *Enlarging the caches or increasing the frequency of the medium and small cores does not affect the general observation that a homogeneous multi-core with big SMT cores is close to optimal.*

### 8.2 Higher memory bandwidth

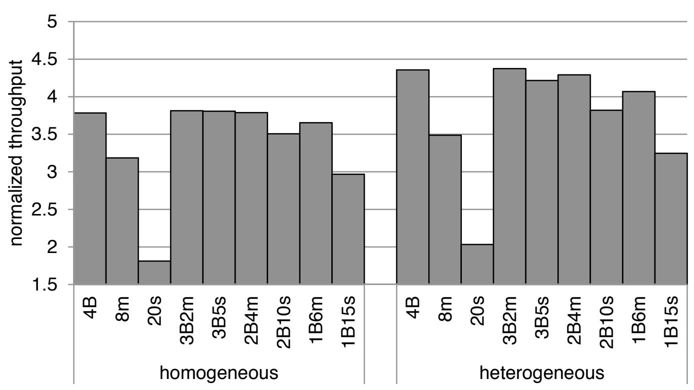
Another decision made in our initial setup was to set the memory bandwidth to 8 GB/s. However, as mentioned before, for some benchmarks, memory bandwidth turns out to be a bottleneck. We therefore now double memory bandwidth to 16 GB/s, see Figure 17. Comparing this Figure to Figures 8 and 11, we observe that performance increases for all configurations, albeit by a small margin. For the homogeneous multi-program workloads, 4B now achieves a 0.8% lower throughput than the optimum (which was 0.6% for 8 GB/s), and a 0.4% lower throughput for the heterogeneous multi-program workloads (used to be 0.5% *higher*). For the multi-threaded programs, considering ROI only, we observe a speedup for 4B that is 2.9% lower than the optimum (which was 1.8% before), and a 1.8% higher speedup when considering the whole program (1.9% before). The programs that were bandwidth-bound in the 8 GB/s setup now achieve better performance across all configurations. These memory-bound benchmarks especially benefit from SMT, more so than compute-bound programs [21]. Hence, our conclusion:

**Finding #11:** *Even under high available memory bandwidths does the performance of a homogeneous design with big SMT cores remain close to the heterogeneous configurations.*

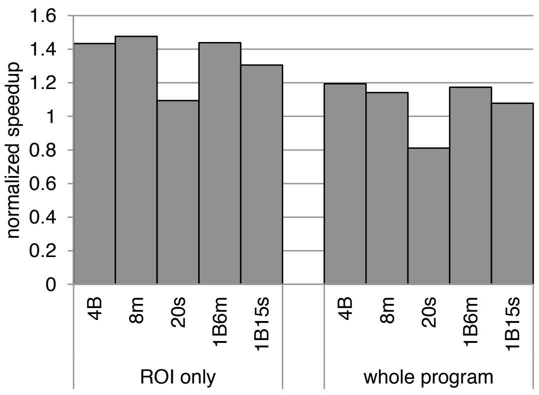
## 9. Related Work

Olukotun et al. [23] make the case for multi-core processing. By comparing an aggressive single-core processor (6-wide out-of-order) and a dual-core processor consisting of 2-wide out-of-order cores, they found that parallelized applications with limited parallelism achieve comparable performance on both architectures, and that applications with large amount of coarse-grained parallelism achieve significantly better performance on the dual-core.

Kumar et al. [19] argue that a single-ISA heterogeneous multi-core processor covers a spectrum of workloads better than a conventional multi-core processor, providing good single-thread performance when thread-level parallelism is low, and high throughput when thread-level parallelism is high. Our results confirm this finding: the heterogeneous multi-core configurations achieve better overall performance compared to 4B across the broad range of active thread counts when SMT is not enabled. However, Kumar et al. did not consider and compare against a homogeneous multi-core with big SMT cores, which we find to achieve a level of performance that is competitive to a heterogeneous design



(a) multi-program



(b) multi-threaded

**Figure 17.** Performance numbers assuming 16 GB/s memory bandwidth.

under varying degrees of thread-level parallelism, while being less costly to design and verify.

Ipek et al. [11] and Kim et al. [17] propose to fuse small cores to form bigger cores when there are few active threads. By doing so, the multi-core processor becomes more dynamic and can more easily adapt to software diversity. Our results indicate that similar performance benefits can be achieved through the opposite mechanism: instead of fusing small cores to form a big core when there are few active threads, one could schedule threads across big SMT cores (and have few active SMT threads per core) in a homogeneous multi-core. Khubaib et al. [16] build on a similar insight when proposing MorphCore, an aggressive out-of-order core with 2-way SMT that can morph into an energy-efficient 8-way SMT in-order core. The idea is to switch between the two modes of operation depending on the amount of available thread-level parallelism: with few (one or two) active threads, the core runs in out-of-order mode, and switches to in-order SMT with more active threads. Whereas Khubaib et al. focus on the proposal of an energy-efficient core design that can switch between out-of-order and wide-SMT in-order operation, the focus of our work is to study the impact of variable thread-level parallelism in the workload, and how this affects multi-core design decisions. More specifically, we consider distributions of active thread counts in multi-program workloads next to multi-threaded workloads to compare homogeneous multi-cores with SMT against heterogeneous and dynamic multi-cores. MorphCore is complementary to our work and can be leveraged to further improve energy-efficiency of the big SMT cores when running multiple SMT threads.

Hill and Marty [10] evaluate the three major multi-core processor architecture paradigms — homogeneous, heterogeneous and dynamic multi-core — and derive high-level insights from Amdahl's Law. Their model did not consider SMT and assumed that software is either sequential or infinitely parallel. One of the results that they obtain is that heterogeneous multi-cores can achieve better performance than homogeneous multi-cores; also, they find that dynamic processors achieve better performance than heterogeneous multi-cores with identical functions of performance per unit area. While this is true considering the assumptions made, our results show that this is not necessarily the case when the number of available threads varies over time.

A number of papers have explored how to take advantage from heterogeneity to improve multi-threaded application performance. Annavaram et al. [1] propose running sequential portions of a multi-threaded application at a higher power budget, thereby significantly improving performance while remaining within a given power budget. Intel's TurboBoost [25] offers similar functionality by boosting clock frequency. Suleman et al. [29] accelerate the execution of critical sections by exploiting high-performance cores in a heterogeneous multi-core, i.e., a thread that executes a critical section is migrated to a big core in order to reduce serialization time — a technique called Accelerating Critical Sections (ACS). Joao et al. [12] generalize this principle to other types of synchronization bottlenecks, including critical sections, barriers and pipes. All of these approaches exploit the fact that the number of active threads varies over time and leverage heterogeneity to improve performance. This paper suggests that similar performance benefits might potentially be achieved through SMT on a homogeneous multi-core. More specifically, when a thread is executing sequential code (e.g., initialization, critical section, etc.), scheduling it on a single core with the other SMT threads throttled, might achieve similar performance benefits, and does not require migrating (or marshaling [30]) data when a thread is migrated from a small to a big core in ACS.

Li et al. [21] compare the energy-efficiency and thermal characteristics of SMT versus multi-core. They report that, assuming an equal area budget, SMT is more energy-efficient than multi-core for memory-intensive workloads; the inverse is true for compute-intensive workloads. Kumar et al. [18] exploit dynamic time-varying application behavior to schedule applications on the most energy-efficient core in a heterogeneous multi-core, and they report substantial energy savings compared to a homogeneous multi-core. In contrast to this prior work, we explore multi-core configurations under variable thread-level parallelism conditions.

## 10. Conclusion

The number of active threads varies over time in today's computer systems. This has been observed across many application domains, ranging from multi-program systems, desktop applications, datacenter servers, and even multi-threaded applications. This paper studied how varying degrees of thread-level parallelism in the workload affect multi-core design decisions. We considered homogeneous, heterogeneous and dynamic multi-cores under an equal power budget, and conclude that a homogeneous multi-core consisting of big SMT cores achieves comparable or slightly better performance compared to heterogeneous multi-cores (both with and without SMT) and dynamic multi-cores. The reason is that a homogeneous multi-core with big SMT cores can better adapt to varying degrees of thread-level parallelism in the workload, and achieves higher per-thread performance at low active thread counts and competitive throughput at high active thread counts. Finally, we also find that heterogeneous multi-cores are (only) slightly more energy-efficient compared to a homogeneous all-big-core configuration with SMT, when power gating idle cores.

The overall conclusion is that, while multi-cores with many small cores, be it homogeneous or heterogeneous architectures, outperform homogeneous multi-cores with big SMT cores at full utilization, the inverse is typically true under variable active thread workload conditions, which makes homogeneous multi-cores with big SMT cores an appealing, cost-effective design point for the variable active threads workloads commonly observed in modern-day systems.

## Acknowledgments

We thank the anonymous reviewers for their valuable and constructive feedback. Stijn Eyerman is a postdoctoral fellow of the Research Foundation–Flanders. 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 at the VSC Flemish Supercomputer Center.

## References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[2] L. A. Barroso and U. Hölzle. The case for energy-proportional systems. *IEEE Computer*, 40:33–37, Dec. 2007.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[4] G. Blake, R. G. Dreslinski, T. N. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 302–313, June 2010.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, Nov. 2011.

[6] K. Du Bois, S. Eyerman, J. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[7] S. Eyerman and L. Eeckhout. System-level performance metrics for multi-program workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[8] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, Sept. 2011.

[9] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, Oct. 1998.

[10] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, July 2008.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 186–197, June 2007.

[12] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[13] M. T. Jones. Inside the Linux scheduler: The latest version of this all-important kernel component improves scalability. http://www.ibm.com/developerworks/linux/library/l-scheduler/index.html, June 2006.

[14] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30:7–15, March/April 2010.

[15] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, Mar. 2007.

[16] K. Khubaib, M. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 305–316, Dec. 2012.

[17] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. Keckler. Composable lightweight processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 381–394, Dec. 2007.

[18] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the ACM/IEEE Annual International Symposium on Microarchitecture (MICRO)*, pages 81–92, Dec. 2003.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 64–75, June 2004.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[21] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–82, Feb. 2005.

[22] NVidia. Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.com/content/PDF/tegra_white_papers/ Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance-v1.1.pdf, 2011.

[23] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, Oct. 1996.

[24] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–26, Sept. 2003.

[25] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32: 20–27, March/April 2012.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[27] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, Nov. 2000.

[28] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 414–425, June 1995.

[29] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multicore architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Mar. 2009.

[30] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 441–450, June 2010.

[31] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 191–202, May 1996.

[32] R. Velasquez, P. Michaud, and A. Seznec. Selecting benchmark combinations for the evaluation of multicore throughput. In *The IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 173–182, Apr. 2013.