The Interaction between Workloads and Architecture in Highly-Parallel Chip Multi-Processors

Oved Izchak

The Interaction between Workloads and Architecture in Highly-Parallel Chip Multi-Processors

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Oved Izchak

Submitted to the Senate of the
Technion – Israel Institute of Technology

Tamuz 5773 Haifa June 2013

Acknowledgements

The research thesis was done under the supervision of Prof. Uri C. Weiser, Prof. Idit Keidar, and Prof. Avinoam Kolodny, in the Faculty of Electrical Engineering.

I am deeply grateful to my three advisors for their wise guidance, inspiration, passion and patience. My appreciation for your care and support for my education and growth cannot be overstated; thank you.

I thank Dr. Zvika Guz for introducing me to this area of research and for his collaboration and contribution to this work.

I thank Prof. Ran Ginosar and Prof. Avi Mendelson for their insights and valuable advice.

I thank Yaniv Ben-Yitzhak for his collaboration and illuminating discussions, in particular about architecture aspects close to the scope of this work and beyond it, which greatly improved my perspective and understanding.

Last but not least, I thank my colleagues in Prof. Idit Keidar's research group: Liat Atzmon-Guz, Ittay Eyal, Dmitri Perelman, Dmitry Basin, Nathaniel Azuelos, Elad Gidron, Eyal Zohar, Mark Silberstein and Stacy Patterson. Collaborating and having discussions with you was pure joy.

Table of Content

List of Figures

List of Tables

Abstract

Highly parallel architectures, such as GPUs or CPUs with vector instructions, require a lot of code tuning to achieve high utilization, i.e., in the order of the theoretical maximum performance. One of the main reasons is that due to technological limitations (e.g., power consumption, power density, availability of instruction level parallelism) highly parallel architectures trade off single-instructions-stream performance for maximum raw-performance. This puts a burden on the workload to provide enough parallelism to keep the architectural resources busy. While some workloads are inherently highly parallel (such as what is known as *embarrassingly parallel*), many interesting, compute-intensive workloads (animation, pattern recognition, ray-tracing) become harder to parallelize as the parallelism degree increases.

In this research we developed a simulator for highly parallel architectures (up to 2048 cores) that can simulate existing parallel benchmarks (any benchmark that runs on the Linux platform) and we use it to study a suite of interesting parallel workloads, the Parsec benchmark suite.

We characterize *parallelism scalability* of each benchmark, namely how the performance scales with the scaling of the architecture's parallelism (core count without overhead). We study another aspect of the Parsec benchmark suite -- shared cache performance (miss-rate) when running with high parallelism degree. We compare the actual performance to an analytical model proposed in the literature.

We find that the inherent parallelism scalability of the various benchmarks in the Parsec benchmark suite varies widely, from as poor as achieving peak performance at 4 threads (ferret) to unlimited, embarrassingly parallel (blackscholes). However the majority of benchmarks do not scale beyond 128 threads.

For cache performance, we find that while for most benchmarks it is compatible with a cache performance analytical model that is proposed in the literature, the performance is highly sensitive to the cache performance and therefore the small differences between the analytical and actual cache performance lead to large differences between analytical

performance estimation based on the analytical cache performance model and the actual performance.

Abbreviations

AMAT	Average Memory Access Time
EPTCS	Effective Per-Thread Cache-Size
LLC	Last Level Cache
LRU	Least Recently Used
IPC	Instructions Per Cycle
SMT	Symmetrical Mutli-Threading
ITC	Inter-thread communication

Chapter 1.
Introduction

1.1. Research overview

Contemporary CPU architecture trend for increasing performance is primarily towards increased parallelism, due to technological challenges such as power dissipation, power consumption, micro-architecture improvements etc. However, parallel architectures require the program to explicitly expose suitable degree of parallelism (e.g., number of threads) to be able to actually utilize the available architecture parallelism. Moreover, the parallelism needs to be exposed for the different type of resources such as utilize cores through threads, utilize cache through data working-set arrangement, utilize interconnect through communication/computation pipelining etc. Underutilization is the result of algorithmic dependencies such as data dependency, inter-task dependency, etc. Often, the situation is in between: part of the time the resources are fully utilized and the rest of the time they are underutilized. Adding resources may increase the performance during full utilization but the extent of the improvement is subject to Amdahl's Law [1], i.e., it depends on the fraction of full utilization periods and may be negligible.

Moreover, whether the architecture's resources are fully utilized and at what times during the program execution depends on the type of resource and on the structure of the program. For example, a program may have a certain data working set at certain times that may or may not fit in cache, depending on the amount of cache in the underlying architecture. If the working set doesn't fit in the cache, then a fast CPU will be underutilized because it can't access data fast enough to utilize its compute power. If the program reduced its working set it could achieve higher CPU utilization. Thus, matching the workload and the architecture can have a dramatic effect on the performance. This is more so with the high-performance architectures which have to tradeoff the amount of resources put into the architecture and technological limitations such as power consumption, DRAM latency, interconnect throughput etc. In this research we study some aspects of parallel benchmark on highly parallel architectures.

There are many aspects of benchmarks that can be measured and studied, at least one for each type of resource. We chose to study the first-order performance limiting factor of a parallel benchmark – its ability to utilize more cores. We study this for a diverse set of benchmarks of the Parsec benchmark suite [2][3]. We find the maximum number of cores that each benchmark can utilize on architecture with unlimited resources, thus we capture the inherent parallelism limitation of the benchmarks.

Another aspect of parallel benchmarks that we study is the effect of high parallelism degree on shared cache performance. With high parallelism degree and assuming independent parallel tasks[1], the nature of the basic assumption that makes cache useful, namely locality-of-reference, changes. At the extreme, when no data is shared among the parallel tasks, the cache is effectively divided between the parallel tasks. This assumption can be used to construct an analytical model for shared cache performance (miss-rate) [4] We extract the actual cache performance from simulation and show that the analytical model for the cache performance is a good first-order approximation, nevertheless it is not ideal for analytical performance studies because the latter can be highly sensitive to the actual cache performance so that even a small mismatch between the analytical and actual cache performance results in a large mismatch between the analytical and actual performance.

1.2. Summary of Contributions

The main contributions of our research are:

- Construct efficient simulator for highly parallel architectures that can run any Linux benchmark.
- Identify and studying the maximal number of cores that each benchmark in the Parsec benchmark suite can utilize.
- Validating an analytical shared cache performance (miss-rate) model for highly parallel architectures.

[1] Task independence is extremely important to minimize algorithmic dependency and therefore is a reasonable assumption for parallel workloads.

1.3. Research Method

Studying the interaction of highly parallel workloads and architectures requires simulating architectures with a parameterized degree of parallelism that scales to high numbers (hundreds and thousands of cores). Moreover, it requires benchmarks that are representative of parallel workloads. We use the benchmarks in the Parsec benchmark suite [3], which includes diverse workloads and provides control over the parallelism degree. For the architecture, we find that full-system simulators are not suitable for our purpose because these do not scale well to the parallelism degrees we want to study and the OS (Operating System) they run (general-purpose OS, typically Linux) does not support such high parallelism degrees. Furthermore, parallel benchmarks are compute-intensive, hence OS kernel space processing is not only unessential to simulate, it actually constitutes noise because of compute resource consumption by background processing (e.g., interrupts, daemons). Therefore, we developed our own simulator that can execute Linux programs, and thus supports any Linux benchmark, and yet simulates only user-space code. We use this simulator to study the various aspects of workloads behavior, both global execution summary statistics and performance counters that reflect the behavior over time.

Chapter 2.
Many-core Architecture Model

2.1. System Model

Our system model is the unified multi-/many-cores model defined in [4], depicted in **Figure 2-1**. It is a shared-memory system with an array of N_{PE} homogenous Processing Elements (PEs or cores) and a memory hierarchy that consists of PE-private caches (traditionally referred to as L1 caches) a shared cache (traditionally referred to as Last-Level-Cache or LLC) of size $S_{\$}$, and main memory (RAM). It is a Symmetrical Multi-Threading (SMT) architecture system [5], i.e., has a register file with room for possibly more thread-contexts than PEs. A scheduler assigns thread-contexts to PEs and changes this assignment according to some scheduling policy (e.g., context-switch when a thread is stalled waiting for data from RAM). It should be noted that when the number of thread-contexts is equal to the number of PEs, the system is not an SMT one so non-SMT systems also map to this model.

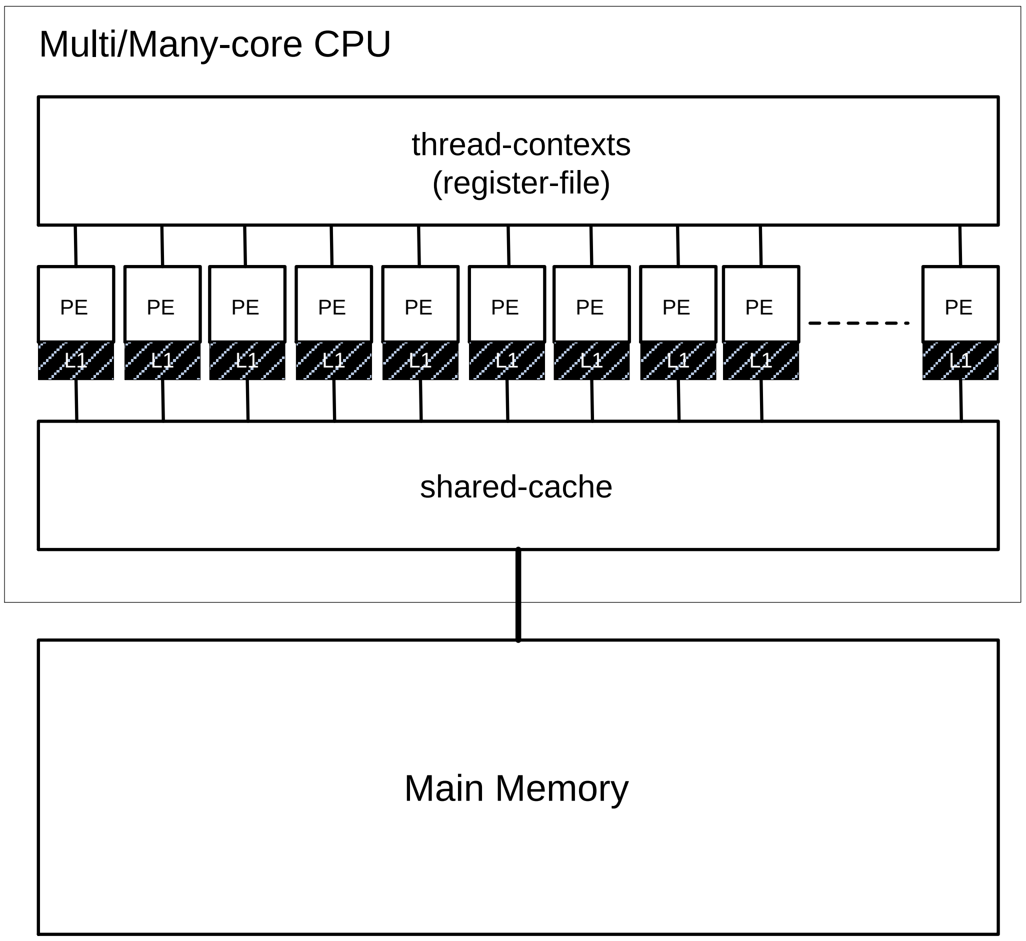


Figure 2-1: Architecture model

The memory hierarchy model is a simple fixed-latency-per-hierarchy-level model with shared cache latency of $t_\$$ and main memory latency of t_m. The private cache latency is modeled indirectly in a workload parameter, discussed later.

The PEs are simple in-order cores with a fixed-latency for the computation part of the instructions denoted CPI_{exe}. A memory access adds the memory-hierarchy latency to the respective instruction latency. **Table 2-1** summarizes the parameters of the hardware architecture part of the model.

Parameter	Description
N_{PE}	Number of PEs (in-order processing elements)
$S_{\$}$	Cache size *[Bytes]*
N_{max}	Maximal number of thread-contexts in the register file
CPI_{exe}	Average number of cycles required to execute an instruction assuming a perfect (zero-latency) memory system *[cycles]*
$t_{\$}$	Cache latency *[cycles]*
t_m	Memory latency *[cycles]*

Table 2-1: System parameters

2.2. Analytical Model

[4] defines an analytical model for the combination of the above system model and synthetic workload. A fundamental assumption of this synthetic model is that the task can be partitioned to any number of *n* sub-tasks, each handled by a separate thread. This assumption effectively ignores data-dependent variability.

The fraction of memory instructions in the synthetic workload's dynamic instructions stream is denoted by r^*_m ($0 \leq r^*_m \leq 1$). Assuming the private cache miss-rate is P^*_{miss}, the fraction of instructions that access the shared cache is $r_m = r^*_m \cdot P^*_{miss}$. In particular, with no private cache (i.e., $P^*_{miss} = 1$) $r_m = r^*_m$.

The shared cache miss-rate depends on the specific workload characteristics (e.g., the working set, amount of data sharing between the threads), the number of threads *n* and the size of the cache $S_\$$. The workload characteristics are captured in the miss-rate function $P_{miss}(S_\$, n)$. **Table 2-2** summarizes the workload parameters.

Parameter	Description
n	Number of threads that execute or are in *running* state (not blocked) concurrently
r_m	Fraction of instructions accessing memory out of the total number of instructions [$0 \le r_m \le 1$]
$P_{miss}(S_\$, n)$	Miss-rate for cache of size $S_\$$ shared by *n* threads

Table 2-2: Workload parameters

In this model the only shared resource is the shared cache. Thus, the shared cache miss-rate function captures the shared resource contention effects of the interaction between the architecture and the workload. This analytical model does not capture the data dependency effects.

The parameters defined in **Table 2-1** and **Table 2-2** are used to analyze expected performance. In this context, for simplicity it is assumed that the workload parameters are fairly static, and do not vary much over time or space (i.e., between different threads of the same application). Therefore, the performance analysis uses average values in the equations below.

With no loss of generality, time is measured in cycles rather than in time units. This saves the need to incorporate the operational frequency into the analysis. Converting a result that is given in cycles units to the respective result in time units is straight-forward.

Given the per-thread shared cache miss-rate function and the cache and memory latencies as defined by the system model, the average number of cycles needed for data access, denoted t_{avg} (sometimes called Average Memory Access Time - *AMAT*) is given by:

(2-1) $$\boldsymbol{t_{avg}(S_\$, n)[Cycles] = \left(1 - P_{miss}(S_\$, n)\right) \cdot t_\$ + P_{miss}(S_\$, n) \cdot t_m}$$

Formulating the average CPI is straight forward:

$$\text{(2-2)} \quad \boldsymbol{CPI_{exe}(S_{\$}, n) = CPI_{exe} + r_m \cdot t_{avg}(S_{\$}, n)}$$

Assuming a thread scheduling policy of context-switch whenever a thread is stalled on a memory access, any given thread needs to stall once every $1/r_m$ instructions on average, and wait until the data it accesses is received from memory. During this stall time, the PE is left unutilized, unless other threads are available to switch-in. The number of threads needed in order to fill a single PE's stall time, i.e., to saturate the PE, is:

$$\text{(2-3)} \quad \boldsymbol{N_{max} = 1 + t_{avg}\frac{r_m}{CPI_{exe}}}$$

With all the PEs saturated, i.e., $n \geq N_{PE} \cdot N_{max}$, each PE executes *$1/CPI_{exe}$ instructions-per-cycle* (IPC) and the aggregate performance of all the PEs is thus N_{PE}/CPI_{exe}. With the PEs not saturated, i.e., $n < N_{PE} \cdot N_{max}$, each thread executes $1/CPI_{avg}(S_{\$}, n)$ instructions-per-cycle so the aggregate performance of all the threads is given by:

$$\text{(2-4)} \quad \mathbf{Performance(S_{\$}, n)[IPC] = \frac{n}{CPI_{avg}(S_{\$}, n)}}$$

Rewriting equation (2-4) in term of the model parameters we get:

$$\text{(2-5)} \quad \mathbf{Performance(S_{\$}, n)[IPC] = \frac{n}{CPI_{exe} + r_m \cdot \left\{\left(1 - \boldsymbol{P_{miss}(S_{\$}, n)}\right) \cdot \boldsymbol{t_{\$}} + \boldsymbol{P_{miss}(S_{\$}, n) \cdot t_m}\right\}}}$$

2.3. Simulation Model

We developed a simulator for the architecture model depicted in section 2.1. The simulated system includes an array of N_{PE} simple cores, with fixed latency per instruction[2]. Memory instructions suffer additional latency incurred by accessing the memory hierarchy, determined by a *memory hierarchy model*.

The memory hierarchy model consists of per-core private caches (L1), a cache that is shared by all the cores (Last-Level-Cache – LLC) and main memory (typically DRAM). Every level of the memory hierarchy is fixed latency and according to where the data is

[2] Including branches – effectively assuming a perfect branch predictor.

found, the latency is added to the latency of the respective instruction. Thus memory-hierarchy contention effects are not modeled (e.g., bandwidth constraints; queuing effects).

There are N_{max} thread-contexts ($N_{max} \geq N_{PE}$) and a scheduler that switches contexts when a thread stalls due to a memory operand that needs to be fetched from Main memory. Context switch takes zero time.

The analytical model depicted in section 2.2 is defined in terms of averages, thus it doesn’t capture the effect of space and time variations. In particular it doesn’t capture data-dependency effects. The simulation does capture the space and time variations effect because it actually executes the program and it does capture data-dependency effects through correctly simulating inter-thread synchronization: when a thread blocks through the OS (typically waiting on synchronization object), it is removed from the simulation scheduling, and once it is unblocked it is re-added. Thus simulation progresses with the blocked threads on one hand not making progress and on the other hand not consuming resources (compute, cache), which exactly matches the effects of the data-dependency.

Using this simulator we can study how real benchmarks behave under the architecture model depicted in section 2.1. Specifically, we can extract both architecture-level and workload-level statistics, with single cycle resolution. Examples for architecture-level statistics are core utilization, average memory access latency, context switch rate etc. Examples for workload-level statistics are instruction mix in the instruction stream (such as the r_m parameter for the analytical model, described in Table 2-2) memory access locality in space and time, inter-thread dependency such as synchronization etc. In particular, we use simulator statistics for extracting the workload-specific model parameters of Table 2-2, which themselves may depend on architecture parameters (which are therefore simulation parameters) such as cache size ($S_\$$ in Table 2-1).

Chapter 3.
The Simulator

In this chapter we present the simulator that we developed to study the behavior of existing benchmarks under the system model depicted in section 2.1. Our simulator executes native Linux benchmarks, and thus allows running standard benchmarks as well as custom benchmarks written in any of the abundant of programming tools that are available for Linux.

The simulator allows easy modifications of architecture parameters (e.g., core count, memory hierarchy configuration, cache configuration, etc.) and collecting comprehensive micro-architecture-level statistics (e.g., core stalls, cache lines utilization).

3.1. Basic requirements

Our basic requirements for our simulator are that it runs Linux programs and that it simulates only the user-space execution. We argue that for the purpose of simulating benchmarks that target highly parallel architectures, simulating the kernel-space code execution is not only unnecessary, it’s undesirable. In this section we explain the motivation behind these basic requirements.

Benchmarks that are not tied to a specific architecture are typically implemented for general-purpose operating systems, primarily Linux (but sometimes also for Windows and/or UNIX) to make them usable across many environments. In particular, there is an abundance of software development tools for general-purpose OSs, which facilitates implementing custom benchmarks as well as tweaking existing ones. Therefore, we target our simulator to be able to execute benchmarks implemented for Linux.

A straightforward approach for simulating a program implemented for a specific OS is a full system simulator [6][7][8], i.e., simulating an entire computer system and running a real-world operating system. However, in general, highly parallel architectures target compute intensive workloads. In fact, contemporary existing highly parallel architectures are typically used as accelerators [9][10], with no OS at all and no direct I/O capabilities[3]. Thus, simulating OS I/O services is not essential for throughput benchmarks.

Even when a throughput workload is running on a full-fledged OS, maximizing compute throughput typically involve preventing context switches to avoid the associated overhead. This is achieved by spawning no more threads than there are hardware thread-contexts. With this and under an assumption that no other programs are running on the system, no context-switch is needed, effectively neutralizing the OS scheduler. Thus, although the benchmarks we target for our simulator are implemented for a general-purpose OS (Linux), we assume that no OS-level scheduling is taking place so it's not necessary to simulate the OS scheduler.

Moreover, general-purpose OSs (such as Linux) perform background maintenance operations (interrupts, daemons) which trigger context-switches and consume core cycles, thus "polluting" the simulation with unrelated and non-deterministic computation resource consumption. As an example, in [11] a full-system simulator was used to gather traces for offline trace-driven simulation but the OS computation was filtered out of the trace because it was undesirable in the simulation.

Not all effects of OS services on throughput benchmarks simulation are undesirable. The effects of *Inter-Thread Communication* (ITC) such as locking, are essential for capturing the inter-thread data dependency, which can be a principal performance limiting factor in parallel workloads. Therefore, we target our simulator to be execution-driven (as opposed to trace-driven) to capture this important aspect of throughput benchmarks. However, we do argue that it's desirable to factor out the OS computation of ITC services in order to capture the workload's inherent data-dependency effects on the performance regardless of specific OS implementation of ITC services. ITC implementations are subject to various optimization tradeoffs that may be configurable or evolve between versions of the OS. Moreover, highly parallel architectures may implement ITC primitives in

[3] In practice the benchmarks do use the OS I/O services, but primarily for setup (e.g., taking execution parameters) and teardown (e.g., reporting results) but the core of the benchmark doesn't use OS I/O services.

hardware. So we would like to preserve the effects of ITC on the simulated benchmarks without simulating their implementation.

To summarize, we want our simulator to simulate only the user-space code of the workloads, effectively assuming that all OS services (I/O or otherwise) do not consume computation resources but still preserve ITC effects.

An existing simulator that scales to the degree of parallelism that we need is Graphite [12]. However Graphite is optimized for low single-simulation latency and for that it exploits distributed setting. However, to facilitate distributed simulation, Graphite employs relaxed synchronization model. For our study, accurate synchronization is vital because synchronization can be a principal parallelism-limiting factor.

It should be noted that the simulation captures shared-resource contention effects, namely the memory hierarchy, similar to the analytical model, but it also captures the data-dependency effects, by simulating the ITC effects, which the analytical model doesn't capture.

3.2. Architecture

We implement user-space simulation by means of binary instrumentation. Binary instrumentation is the process of in-memory modification of a program while it executes, allowing interleaving of the program's code with custom instrumentation code that can observe and/or affect the original code's behavior. Binary-instrumentation-based simulation has the additional benefit of not requiring the simulated application to be implemented using a specific language or framework and doesn't require re-compilation. Among other things, this allows simulating applications that execute under a runtime-environment program such as the Java Virtual Machine. Our simulator is built on the Pin binary instrumentation framework from Intel [13].

3.2.1. Functional execution scheduling

Since with binary instrumentation the application's instructions are functionally executed on a physical processor, the physical processor provides the ISA functional-model. Thus, simulation requires the program instructions to be scheduled to the physical processor in the order imposed by the timing model of the simulated architecture. For example, when two threads contend on changing the same memory location (typically using atomic Read-Modify-Write instructions), the one that would execute first according to the timing model should be scheduled first to the physical processor.

Simulation is implemented through binary instrumentation by having the binary instrumentation framework instrument the benchmark program with simulator code that:

- observes the program’s dynamic instructions streams (threads)
- executes a timing model of the simulated architecture on the observed instructions streams
- schedules the execution continuation according to the outcome of the timing model execution.

This is depicted in Figure 3-1. The instruction streams of the application threads are instrumented with callback operations into the simulator’s code. These provide the simulator with all the information about the subsequent instruction of each instruction stream, such as whether it’s a memory instruction or not, and if it is, whether it is a read or write operation, the referenced address, the operand size, etc. Therefore, as the code executes the simulator gets to see each instruction and its operands before it executes.

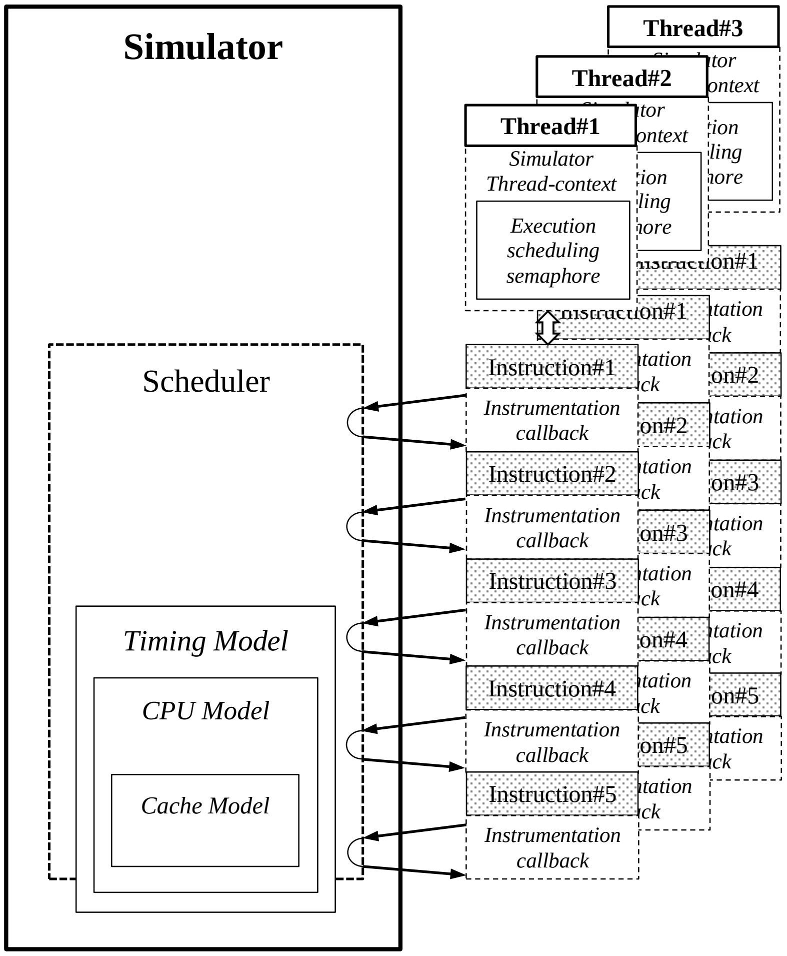


Figure 3-1: Application-Simulator interaction

To achieve the effect of execution-driven simulation, the instructions of the different threads must be executed in the relative order that the simulated architecture imposes. This means that the next instruction of a given thread must be allowed to execute only when it is determined that all the next instructions of all other running threads are ordered after it according to the simulated architecture's timing model. In other words, given a set of next instructions of all running threads, the next instruction to schedule for functional execution is the one that is going to complete first according to the timing model. Thus, given the set of next instructions of all running threads, the simulator executes the timing model until it determines which of these instructions will be completed first. Then it lets the respective thread execute this instruction. It then gets notified about the subsequent instruction of that thread, and it again has a set of next instructions of all running threads. It continues the execution of the timing model from where it left off and determines the next instruction that will be completed, and so on.

Controlling the next instruction execution requires the ability to suspend and resume application threads. This is achieved through a per-thread functional execution scheduling semaphore that is maintained in a per-thread-context block. When a callback is called with the information about the next instruction, the instruction is added to the current set of next instructions and the thread is suspended by waiting on its respective semaphore. When the next instruction to execute is determined from the timing model, the instruction is allowed to functionally execute by signaling the respective thread’s semaphore, which make the callback return and the next instruction to execute.

3.2.2. Inter-thread communication effects preservation

To further maintain execution-driven simulation, the ITC effects on the execution must be preserved. There are two main forms of ITCs: one that involves the OS and one that doesn’t.

In ITC that involves the OS a thread makes a *syscall* (system call into the OS kernel) that blocks until another thread makes a complementary syscall that makes the former syscall return. When the blocking syscall is allowed to execute, the respective thread will get blocked inside the syscall. For the syscall to return, the other running threads need to be allowed to make progress in order for the complementary syscall to be reached. During this time the blocked thread cannot make progress. To reflect this, a thread that is about to execute a syscall is removed from the set of currently running threads. This allows simulation scheduling to continue because the timing model no longer expects to have a next instruction from that thread to participate in the timing model execution. When the complementary syscall is executed, the blocking syscall returns, which causes the instrumentation callback following the syscall to be called and the thread is added back to the set of running threads and it again participates in timing model execution and functional execution scheduling.

Instead of distinguishing between blocking and non-blocking syscalls to determine whether to exclude a thread from the set of running threads or not, we treat all syscalls as blocking. If the syscall is not really blocking, it will return quickly and the thread will be added back to the set. Under the assumption that non-ITC syscalls are rarely used in throughput workloads, the treatment of all syscalls as ITC should have negligible effect on simulation results.

The other form of ITC doesn't involve the OS, i.e., user-space-only ITC. Such ITC necessarily uses shared variables for the communication. The effects of such ITC are implicitly maintained by the user-space-only simulation as described above. The canonical example for user-space-only ITC is a spin-lock mutex [14]. A possible straight-forward implementation of spin-lock mutex is the following:

```
subroutine Acquire(Mutex)
    while (CAS(Mutex.state, FREE, OWNED) = OWNED)
    end while
End subroutine
subroutine Release(Mutex)
    Mutex.state ← FREE
    MemoryFence
end subroutine
```

Since instructions are executed in the order imposed by the timing model, the thread that would execute the acquisition CAS first in the simulated architecture will be the one that is scheduled first by the simulator's functional execution scheduler and therefore end up owning the mutex first. Until the mutex is released all other threads will be spinning inside the Acquire subroutine and thus will not make computational progress, which is exactly the inter-thread data dependency effect that the simulator should capture.

3.2.3. Achieving execution-driven simulation

The timing model part of our simulator is effectively a trace-driven simulator – its input is the instruction traces of the application threads. It so happens that the trace is of a single instruction from each thread. Once an instruction is executed, and therefore removed from the trace, the following instruction constitutes the next single instruction trace of that thread and again the input of the timing model is a trace of a single instruction from each thread. The online feedback from this trace-driven timing model to the trace collecting mechanism by mean of binary instrumentation callbacks makes our simulator effectively an execution-driven simulator.

The timing simulation is implemented using classical event-driven simulation design. From an architecture point of view, this allows arbitrarily detailed simulation, e.g., on-chip network routing, cache-coherency protocols, DRAM controller etc. This also allows trading off accuracy for simulation speed through a simplified timing model, e.g., fixed latencies memory hierarchy.

3.3. Implementation

3.3.1. Reducing context switches

Following from the description in section 3.2.1, when two consecutively scheduled workload instructions are from different threads, there is at least one OS-level context switch because after the first instruction executes, the callback blocks on the semaphore and then the second instruction's thread semaphore is signaled and the respective thread unblocks. The more threads are running, the bigger the chance for any pair of consecutively executed instructions to come from different threads. So the number of context switches is in the order of the total number of executed instructions. This is a huge performance overhead.

A key observation here is that in order to preserve the effect of the simulated architecture's timing on the computation, it is not necessary to functionally execute all the instructions in the global order that is imposed by the timing model – it suffices that the functional execution order maintains the relative execution order. In particular, the functional execution of a single thread is not affected by other threads if there are no shared operands. For example, suppose a thread T_A executes a computation sequence that involves only Register operands and no memory operand. The results of this sequence cannot be affected by computations of other threads because other threads cannot change any of T_A's registers and therefore cannot change any of its operands. Thus, the results are not affected by whether other threads' computations are executed before, during or after T_A's computation. In other words, this T_A's computation sequence is unordered relative to other threads.

Similarly to register only operands sequences, if it can be shown that a computation sequence that uses memory operands do not share these memory operands with other threads, then such a sequence is also not affected by the execution of other threads and therefore is unordered relative to other threads. With Linux supporting processors with relaxed memory consistency models [15], i.e., modifications to memory locations may be observed in different order by different threads, the program must use memory fences to make any visibility order guarantees. While in theory different threads may observe fences from different other threads in different order (therefore also in non-real-time order), a simple fence ordering scheme that maintains the appropriate semantics is global real-time order, i.e., all fences are observed by all threads in the same order that also reflect their real-time order.

The order of memory modifications is defined relative to memory fences in their respective thread as depicted in Figure 3-2. It shows the real time execution of two threads T_A^1 and T_A^2. The instruction streams are segmented on memory fence boundaries – the thick black lines denote memory fence instructions. S_1^1 and S_2^1 are consecutive segments of T_A^1. Similarly S_1^2 and S_2^2 are consecutive segments of T_A^2. The instructions in S_1^1 are unordered relative to S_1^2 – each may or may not see modifications made to shared variables by the other. The same is true for S_1^1 and S_2^2. However, it is guaranteed that S_2^1 sees the modifications made by S_1^2 because S_2^1 is ordered after the fence at t_4, which is ordered after the fence in t_3 (because fences are globally ordered), which in turn is ordered after S_1^2. So a segment S_m^i is guaranteed to see the changes made by a segment in another thread S_n^j IFF the memory fence preceding S_m^i in T_A^i is ordered after the memory fence succeeding S_n^j in T_A^j. Otherwise the segments are unordered. Therefore, correct programs must ensure that there are no accesses with undefined relative ordering to the same operands by multiple threads because otherwise, the same initial program state may produce different results in different runs.

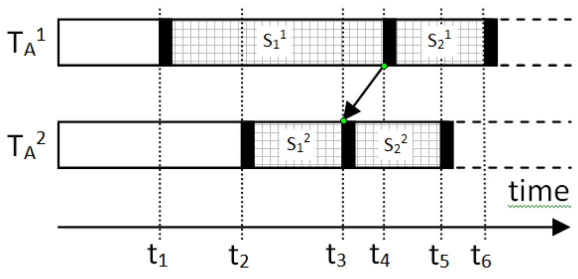
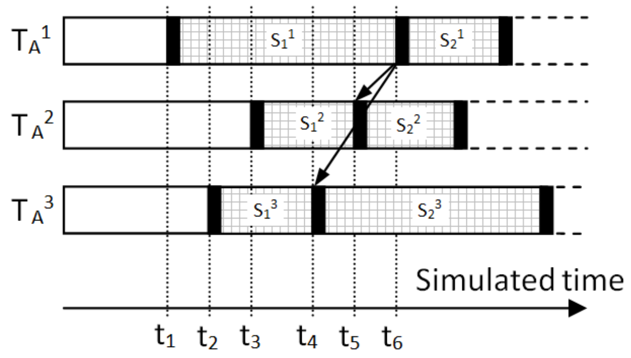
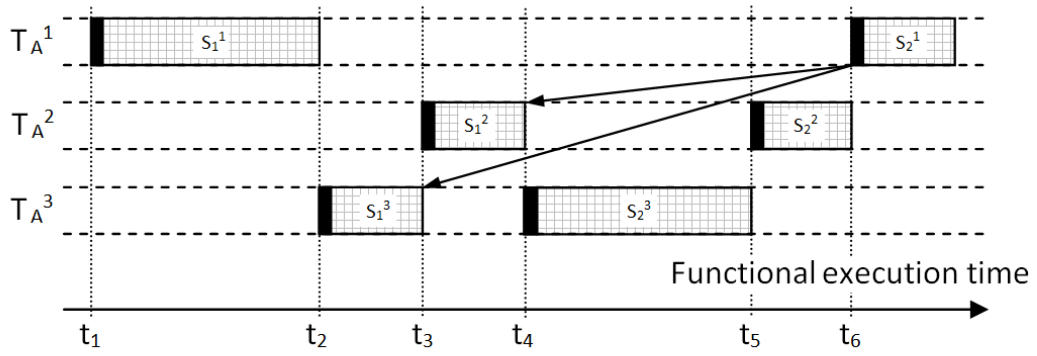


Figure 3-2: Weak memory consistency-model

Whether unordered segments see the memory modifications of each other is undefined. Therefore, they cannot have common operands. Thus, unordered segments can be functionally executed in any order with no change of semantics, i.e., independent of their exact timing according to the timing model. To maintain correct execution semantics it suffices that ordered segments are functionally executed in the right order according to the simulated architecture timing model and for that it suffices that only memory fences are executed exactly according to the timing model. This allows functionally executing instructions of a single thread atomically (i.e., continuously) without considering the timing model as long as no memory fence is encountered. Once a memory fence is encountered, its functional execution must be delayed until all memory fences that precede it according to the timing model have been executed so that subsequent instructions get to see the functional results of other threads' segments that are ordered before them. This is illustrated in **Figure 3-3**. It shows 3 instructions streams of 3 threads: T_A^1, T_A^2 and T_A^3. The small black rectangles denote instructions that are memory fences. **Figure 3-3(a)** shows the functional execution timing in the simulated architecture. The arrows show order dependency, from a segment that is ordered after to the segments that are ordered before it. **Figure 3-3(b)** shows functionally equivalent execution: memory fences are executed in the same order as in **Figure 3-3(a)** and segments that are ordered execute in the right order.



(a) Simulated architecture execution timing



(b) Valid observed functional execution of (a) under week memory consistency model

Figure 3-3: Functional execution scheduling

Segment execution includes trace collection. The timing model still works with one instruction trace per thread but after an instruction is determined to be completed, the next instruction for that thread is taken from the respective thread's trace, if it's not empty. If it is empty, the next segment is allowed to execute and a new trace is collected. Now again there are non-empty traces for all the running threads and the timing model execution can continue.

3.3.2. Timing model execution thread

Figure 3-4 depicts how the functional-execution scheduling is combined with timing model execution to achieve the effect of execution-driven simulation. When an application code segment executes, a trace is collected. Timing simulation takes place when a non-empty trace is available for all running threads. During timing simulation all the threads are blocked on their execution scheduling semaphore. The exhaustion of any of the per-thread traces implies that the next instruction in that thread is next in the global order. At this point timing simulation is suspended and functional execution of the thread whose trace was exhausted is resumed and a new trace is collected, up to the next fence (excluding). Now again there are non-empty traces available for all running threads and again timing simulation can take place.

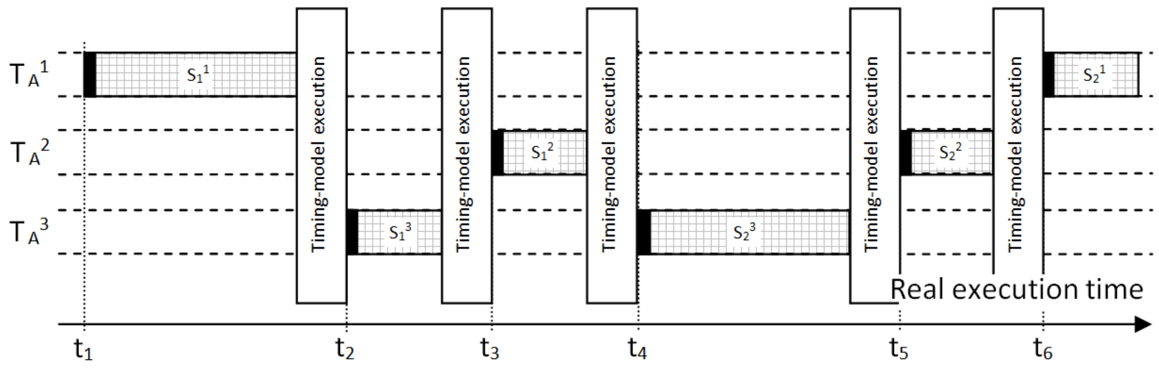


Figure 3-4: The combination of functional and timing-model execution, resulting effectively an execution-driven simulation

To summarize, the simulator alternates between two execution phases:

I. Functional execution and Trace-collection

II. Timing-simulation

In phase I, the workload's code executes on the physical processor (functional execution) and the instrumentation code collects trace by observing the instructions stream. Once traces are collected for all running threads, simulation changes to phase II by suspending the functional execution (all threads are blocked on their respective execution scheduling Semaphore) and executing the timing simulation. Once the timing simulation exhausts any of the thread traces, simulation changes back to Phase I by suspending the timing simulation and resuming the execution of the thread whose trace was exhausted by signaling its execution scheduling semaphore. After the resumed thread collects a new trace, once again traces are available for all threads and timing simulation can continue and so on.

Executing the timing model requires an execution thread-context. One possible approach is to have a dedicated thread T_E. That thread would determine the next instruction to execute, unblock the respective thread T_A^i and wait for it to complete collecting a new trace for that thread. When trace collection completes, the callback of that thread will make the trace available to T_E, unblock it and then block on its scheduling semaphore. This is depicted in Figure 3-5.

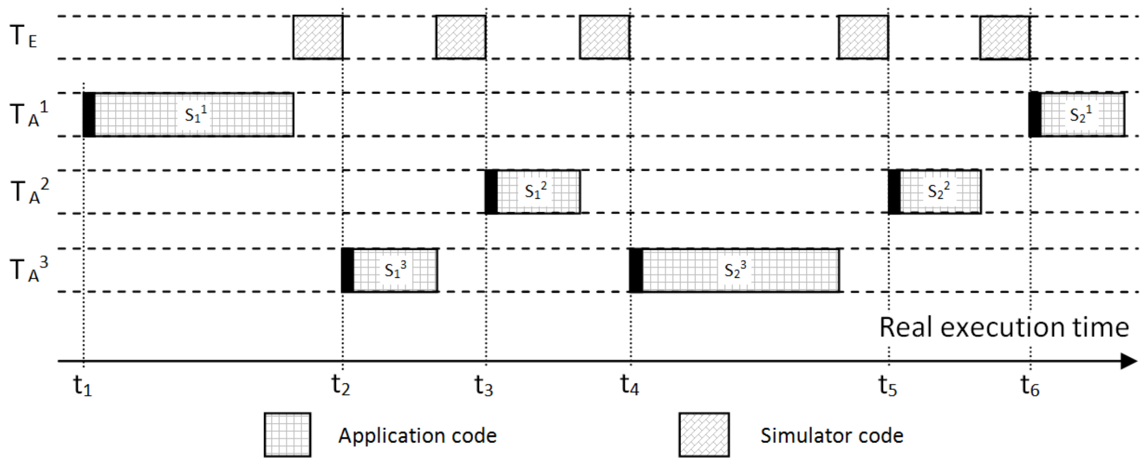


Figure 3-5 Execution scheduling with a dedicated thread for timing model execution (T_E)

This approach implies that for every executed segment there will be two context switches: one from T_E to T_A^i after T_A^i has been determined from the timing model and one from T_A^i to T_E once trace collection for T_A^i completes.

It should be noted that in this approach the execution of T_E is completely serialized with the execution of the application's threads – when one executes all the others are blocked. Therefore, we exploit this fact to save one context-switch per-segment by executing the timing model in the application thread within the callback. This is depicted in Figure 3-6.

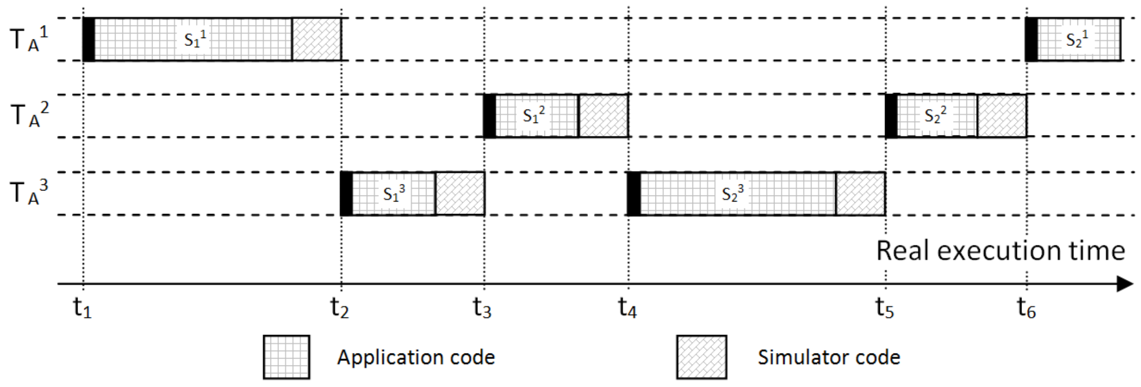


Figure 3-6 Execution scheduling without a dedicated thread for timing model execution

3.3.3. Limiting the size of a segment

We saw that partitioning instruction streams into segments along memory fences and executing the segments atomically is consistent with weak memory consistency model and also allows incurring only one context switch per segment execution. However, a thread may execute an unbounded number of instructions between memory fences, implying that segments lengths are unbounded which in turn implies that trace lengths are unbounded. Long traces increase the memory working-set, potentially incurring significant overhead (e.g., reduces cache utilization, thrashes physical memory). Also, the marginal gain of amortizing the context-switch overhead across long segments diminishes quickly – for example the gain in amortizing the context switch overhead across segments of 20 instructions instead of 10 instructions is much larger than the gain in amortizing across 1010 instructions instead of 1000 instructions. Therefore, it is beneficial to limit the segment length (and therefore the trace length) – it avoids the overhead of larger memory working set on the expense of negligible context-switch overhead due to what could have been a longer segment.

It is important to note that in general, the segment-based functional execution scheduling ensures that first instructions in each segment are executed in the global order that is dictated by the timing model (which is not the case with instructions that are not first in a segment). Thus, instructions that must execute in the correct global order, such as memory fences, must be first in an execution segment. However, any instruction can be executed in the correct global without affecting correctness. So a segment can be made arbitrarily short without affecting correctness, as long as memory fences are always first in a segment. In particular, a segment can be limited to 1 instruction, thus making all instructions first in a segment and hence all instructions are executed in the correct global order, as would be the case if we didn't employ longer traces. This also allows suspending trace collection conservatively, i.e., if an operation may be a fence but not necessarily, e.g., a syscall.

The actual trace length limit is a parameter of the simulator.

3.3.4. Simulated architecture software scheduling

As explained in section 3.1, it is assumed that a throughput workload will not spawn more threads than there are hardware thread-contexts in the simulated architecture and therefore there is no need for a software scheduler, i.e., software-managed context-switching, a software scheduler that suspends a running thread to free its hardware thread-context and assigns another thread to that thread-context. However, in practice a workload may have more active threads than hardware thread-context even if it doesn't spawn more worker thread than there are thread-contexts. For example, a workload may have a main thread than spawn the worker threads and then waits for them to terminate. If all the worker threads start running before the main thread reaches the blocking step, there are more active threads in the program than there are hardware thread-contexts. The simulator handles this by implementing a simple cooperative-multitasking software scheduler i.e., a scheduler that holds aside threads if there is no vacant hardware thread-context and map them to a hardware thread-context when one becomes available, i.e., by a running thread getting blocked on an OS syscall. Thus, in the above workload example where the number of running threads is one more than there are hardware thread-contexts, either the main thread is assigned to a hardware thread-context or it is not. If it is, then one of the worker threads cannot be mapped to a hardware thread-context and therefore the main thread can make progress but one of the worker threads cannot. The main thread will quickly get to block waiting for the worker threads to finish and hence will free its thread-context and the cooperative scheduler will assign the last worker thread to that thread-context. Now only the worker threads are active and therefore all are assigned to thread-contexts, all run in parallel and no further context switch is needed even if any of them blocks. If the main thread is not assigned to a thread-context (for example, because it blocked on something else and then unblocked before blocking on waiting for the worker threads to finish) then all the worker threads are assigned to hardware thread-contexts and they all run in parallel until one of them blocks or terminates. When one blocks or terminates, its thread-context is freed and the cooperative scheduler assigns the main thread to it. Again the main thread will quickly get to the stage that it is blocked on the worker threads and again only worker threads are active so there are at most as many active threads as there are thread-contexts and the worker threads run in parallel until termination. Thus, in either case the worker threads end up running in parallel except for brief period it takes for the main thread to block after spawning all the worker threads.

It should be noted that this software cooperative scheduler graceful handling of the number of active threads exceeding the number of hardware thread-contexts is not limited to exceeding only by one. It can be easily seen that exceeding by more than one is handled equally gracefully, as long as the period of excessive active threads is very small relative to the runtime of the worker threads.

3.4. Summary

To summarize, our simulator executes Linux programs, simulating only the user-space code. It is oriented to throughput workloads, assuming no use of OS I/O or scheduling services. The functional execution is provided by the physical processor and the timing simulation is provided by an arbitrarily detailed timing model that is implemented using event-driven architecture.

Chapter 4. Benchmarks parallelism scalability study

4.1. Introduction

In this chapter we use our simulator to study the inherent parallelism scalability of a widely used parallel benchmarks: the Parsec benchmark suite [3]. What we would like to capture is the benchmarks' inherent limitations in exploiting highly parallel architectures. We capture this through the performance as a function of the degree of parallelism that is available in the underlying architecture (the number of Processing elements or Cores).

To capture the parallelism inherent in a benchmark's algorithm, we use architecture with no inherent parallelism limitation in the architecture itself, such as bandwidth limits or memory latency. Therefore, for the parallelism scalability study we use architecture with a perfect memory system – no cache and every memory access incurs a latency of 1 cycle. We maintain a latency of 1 rather than 0 cycles as an expression of the fact that memory instructions are inherently slower than non-memory instructions.

The metric we use for performance is the average *instructions-per-cycle*. This metric is more appropriate than the conventional *total execution-time* metric because the latter allows meaningful comparison only when the size of the problem is constant across different parallelism degrees. Such parallelism model is compatible with the model used by Amdahl's Law [1]. However, a program may adjust the size of the problem according to the execution parameters such as the parallelism degree. Such parallelism model is compatible with Gustafson Law [16]. Indeed some benchmarks in the Parsec benchmark suite adjust the problem size to the parallelism degree. The IPC metric effectively normalizes the performance over the problem size. It should be noted that this assumes the program does not include speculative computation or else the IPC metric does not reflect the performance – an execution with higher degree of parallelism and higher IPC may have larger portion of its speculative computation discarded than an execution with lower degree of parallelism and lower IPC so that the latter completes faster, thus having higher performance while having lower IPC. The benchmarks in the Parsec benchmark suite do not include speculative computation.

In the Parsec benchmark suite, the degree of parallelism is expressed by the number of *worker-threads*, i.e. the number of threads the program spawns (not all worker-threads necessarily perform computation throughout the program execution, e.g., a thread may block waiting for other threads). The number of worker threads is a runtime parameter of the benchmark. We use simulated architecture that has at least as much cores as there are worker-threads so that core-to-thread allocation does not constitute a parallelism limiting factor.

Table 4-1 summarizes the parameters of the architecture model that is used to study the parallelism parameters.

Parameter	Description
N_{PE}	>= #worker-threads
$S_{\$}$	0 (no cache)
N_{max}	N_{PE}
CPI_{exe}	1 *[cycles]*
$t_{\$}$	Not applicable (no cache)
t_m	1 *[cycles]*

Table 4-1: Model parameters for the parallelism scalability study

With no algorithmic speculative computation, there are two performance limiting factors (i.e., factors that reduce the IPC):

1. Memory latency
   This reduces IPC through incurring extra latency. The degree of IPC reduction depends on the mix of memory and non-memory instructions. This is captured by the workload model parameter r_m, as described in 2.2.

2. Inter-thread synchronization

   This reduces IPC through preventing a thread from executing any instructions while waiting for another thread to complete some computation.

The Inter-thread synchronization reflects the ITC inherent in the workload. Inter-thread synchronization is known to be a principal performance limiting factor for highly parallel architectures, from GPUs to super-computers. The effect of inter-thread synchronization can be visualized through graph of the number of *running threads* (as opposed to blocked threads) over time. For example, suppose a parallel algorithm involves each thread performing multiple iterations where the iterations of different threads need to execute in lock-step. This requires that at the end of every iteration all threads wait until all other threads complete the respective iteration. This type of synchronization is typically achieved using a Barrier synchronization object [17]. Further suppose that the time it takes a thread to execute an iteration is distributes uniformly between t_n and t_x. The number of active threads over time for this algorithm is depicted in **Figure 4-1**. From the beginning of an iteration t_0 until t_0+t_n the number of active thread is equal to the number of worker threads and from there until t_0+t_x the number decrease linearly down to 1 (the last one to complete the iteration), at which time the next iteration can start and all the waiting threads are unblocked.

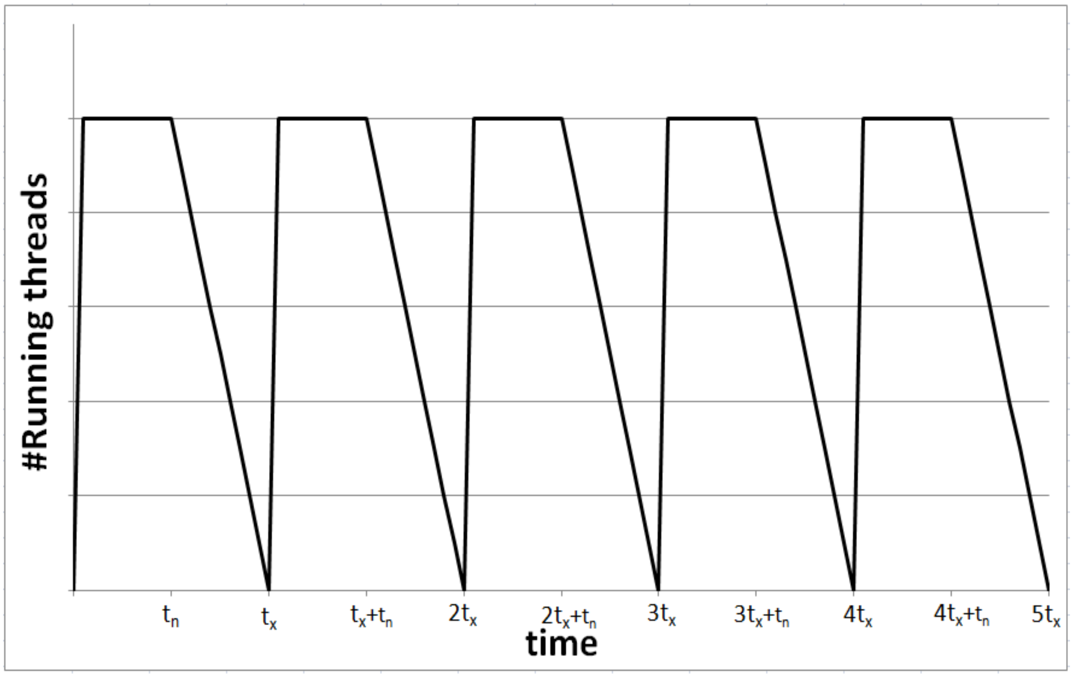


Figure 4-1: thread-context occupancy illustration

For a given number of worker-threads and in the absence of speculative computation, maximum performance is achieved when all the threads are running throughout the program execution. The performance in this case is provided by formula (2-5) in the analytical model (section 2.2). The fact that the architecture that is used to study the parallelism scalability has no cache is equivalent to $P_{miss}(S_{\$},n)=1$. Therefore, the performance formula (2-5) becomes:

(4-1) $$\mathbf{Performance_{max}(n)[IPC]} = \frac{\mathbf{n}}{\mathbf{CPI_{exe}+r_m \cdot t_m}}$$

n is the number of worker-threads; CPI_{exe} and t_m are described in Table 4-1; r_m is extracted from the simulation.

A workload is said to have *perfect parallelism scalability* over a range of worker-threads count when its actual performance is equal to the maximum performance as depicted in (4-1) over that range. Such a workload can fully utilize as much parallelism in the range as available in underlying architecture.

In our study we show the actual performance vs. the number of worker-threads of the actual and maximum performance of the various benchmarks in the Parsec benchmark suite. For the ones that do not exhibit perfect parallelism scalability, i.e., limited by inter-thread synchronization, we show the inter-thread synchronization effects using the graph of running threads over time for several worker-threads counts.

4.2. Measurement methodology

Using the simulator described in Chapter 3, we obtain the total number of executed instructions and the total execution time (in cycles) and calculate the IPC. The details of the simulation environment are provided in Appendix: simulation environment.

We measure with the benchmarks running with up to 1984 worker-threads. The upper limit is derived from a limitation of the Pin binary instrumentation framework, which supports programs with up to 2048 threads. Since most of the benchmarks have a control thread in addition to the worker-threads, these benchmarks cannot be instantiated with 2048 worker-threads under Pin. 1984 was chosen because the higher worker-threads count were selected to be multiple of 64 and 1984 is the largest multiple of 64 that is smaller than 2048. To have measurements in reasonable resolution, the worker-threads counts are not spaced evenly across the range. They are spaced more closely in the lower range. So the simulated worker-threads counts are 4-64 in 4 threads interval, 64-128 in 16 threads interval and 128-1984 in 64 threads interval.

Some benchmarks have constraints on the number of worker-threads, such as it must be a power of two. Some do not run properly with order of hundreds and thousands worker-threads. Some spawn multiple threads per worker-thread count so they hit the 2048 threads limit of the Pin binary instrumentation framework with a lower number of worker-threads parameter. We indicate this in the results of the specific benchmark.

The simulations are performed with the "simmedium" data-set unless otherwise indicated in the results of the specific benchmark.

The benchmarks of the Parsec benchmark suite include setup and cleanup steps that are not part of the parallel algorithm, e.g., reading input data from disk to memory, printing summary information etc. To allow excluding these operation from the measurement, the Parsec benchmark suites define *Region of Interest* (ROI), which is the part of the execution that performs the actual parallel algorithm. The ROI start and end are signaled by calls to specific functions. These signals are made available inside the simulator by sensing these calls. To enable this ROI detection by the simulator, the simulator takes the names of the ROI start/end functions as parameters. The ROI start/end notifications inside the simulator are used to reset/sample the simulator's internal performance counters (on ROI begin/end, respectively) so that the startup and cleanup operations are excluded from the simulation statistics.

The benchmarks in the Parsec benchmark suite notify the ROI start just before spawning the worker threads, after preparing the input data-set in memory. However, this means that the thread spawning operation is included in the ROI but this is not really part of the parallel algorithm. Spawning a thread involves some computation that executes in the context of the spawning thread and some thread-startup computation that executes in the context of the newly spawned thread. We'll refer to these as $C_{prepare}$ and $C_{startup}$, respectively. While $C_{startup}$ of different worker threads can execute in parallel because they execute in the context of different threads, $C_{prepare}$ of different worker threads all execute in the context of the spawning thread and hence are serialized. Thus, spawning the worker threads incurs a serial-execution component that is proportional to the number of worker threads; hence its effect increases with the degree of parallelism. To exclude $C_{prepare}$ and $C_{startup}$ from the ROI, our simulator supports an alternate ROI detection mode, different than detecting the calls to the ROI start/end functions. In this mode the ROI begin is implied from all the threads reaching the thread-entry point function, i.e., after all $C_{prepare}$ and $C_{startup}$ computations have finished. For this mechanism the simulator takes 2 parameters: a list of thread entry-point functions[4] and the expected number of threads to enter these functions. To ensure all the parallel computation is included in the ROI, the simulator freezes the threads as they reach the thread-entry function so that they do not execute code until all worker-threads have been spawned and reached the thread entry-function (i.e., after $C_{startup}$). This has the effect of a barrier at the beginning of the thread-entry functions. Indeed some benchmarks have explicit barrier in the beginning of their thread-entry function to achieve exactly that. However, the Simulator's effective barrier has the

[4] some benchmarks have several entry functions, for different types of worker-threads

advantage that it takes zero simulation time to unblock all the threads, as opposed to a barrier in the benchmark itself, which does consumes simulation execution cycles while actually not being part of the parallel algorithm. Therefore, for the purpose of our simulations we removed barriers in the beginning of thread-entry functions. We indicate this in the results of the specific benchmarks.

Complementary to notifying ROI start just before spawning the worker threads, the benchmarks in the Parsec benchmark suite notify ROI end just after all the worker threads terminate, traditionally referred to as *join*-ing the thread handles. Similar to the case of spawning a thread, joining a thread involves thread-cleanup computations that execute in the context of the joined threads, and computations that execute in the contexts of the joining thread. We'll call them $C_{terminate}$, and C_{join}, respectively. While $C_{terminate}$ of different threads can run in parallel, the C_{join} of different threads are serialized in the joining thread. Moreover, worker threads are not necessarily symmetrical with regards to execution time, i.e., some threads may exit sooner than others. This means that the degree of parallelism is reduced towards the end of execution. Like serial execution computation segments, the effect of this "reduced parallelism tail" increases with the increase in the degree of parallelism so we would like to exclude it from the simulation too. This is achieved by the simulator sensing the exit from the thread entry-point functions and using the first exit as the ROI end indicator. This excludes the $C_{terminate}$ and C_{join} of all threads from ROI, as well as the "reduced parallelism tail". While we recognize that theoretically the "reduced parallelism tail" can be a principal performance bottleneck of a parallel workload[5], we assume that even if it is, this is not inherent to the parallel algorithm but rather a consequence of the particular implementation. Under this assumption, the tail is not representative of the benchmark's inherent parallelism so excluding it for the purpose of studying the parallel part is desirable.

[5] Reference to map-reduce handling of "reduced parallelism tail".

4.3. Simulation results

4.3.1. blackscholes

The **blackscholes** benchmark contains a barrier at the beginning of the worker-threads in order to maximize parallelism by making all the threads start only after all have been created and reached the starting point. As described in section 4.2, a barrier in the workload itself introduces a serial phase so for our measurements we removed the barrier from the benchmark and replaced it by the simulator's support for worker-threads start synchronization.

The **blackscholes** benchmark exhibits perfect parallelism scalability, as shown in Figure 4-2 (the curves of the maximum and actual performance overlap).

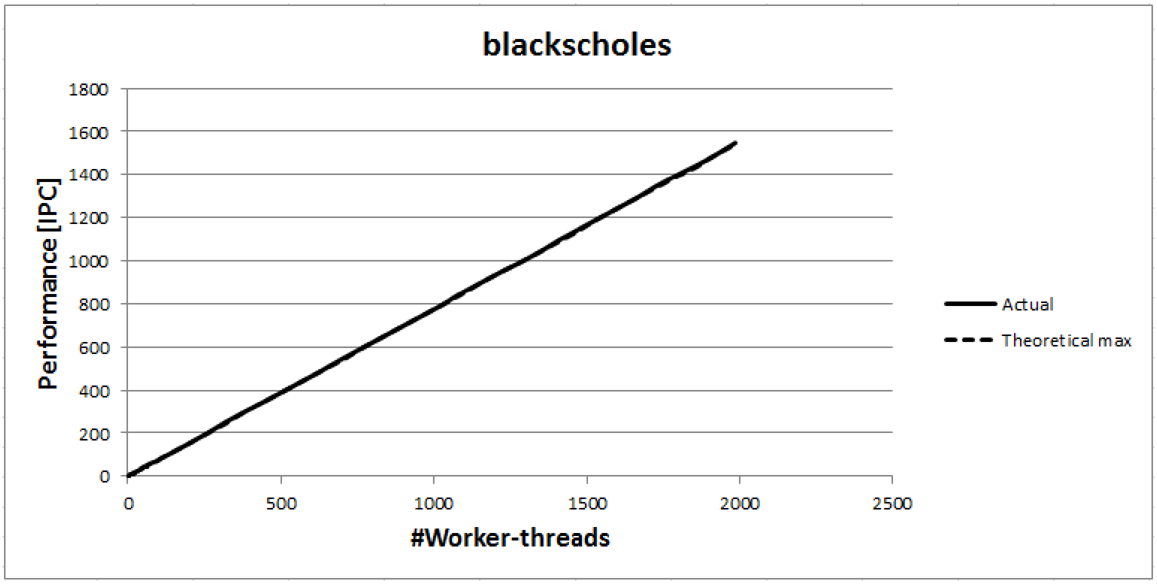


Figure 4-2: Parallelism scalability - blackscholes

The **blackscholes** benchmark is *embarrassingly parallel*, i.e., there is an abundance of the parallel units of work and they are completely independent – the data-set is partitioned evenly between the worker-threads, which operate on them independently, i.e., with no inter-thread synchronization. This is seen in Figure 4-3, which shows that all worker threads are running throughout the benchmark execution, i.e., they never block to synchronize with other threads (we use the 1948 worker-threads execution as a representative – it is the same with smaller number of worker-threads):

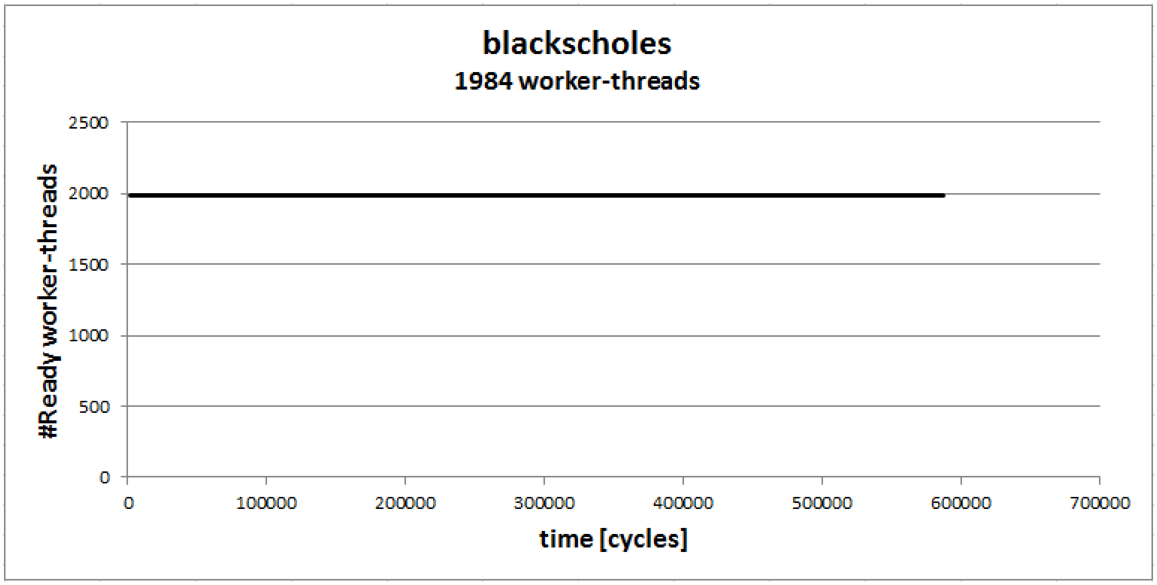


Figure 4-3: Running threads over time - blackscholes

4.3.2. bodytrack

The performance of the **bodytrack** benchmark is shown in Figure 4-4. Figure 4-5 is similar to Figure 4-4 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows good scalability up to ~32 threads and the performance plateaus at ~128 worker-threads.

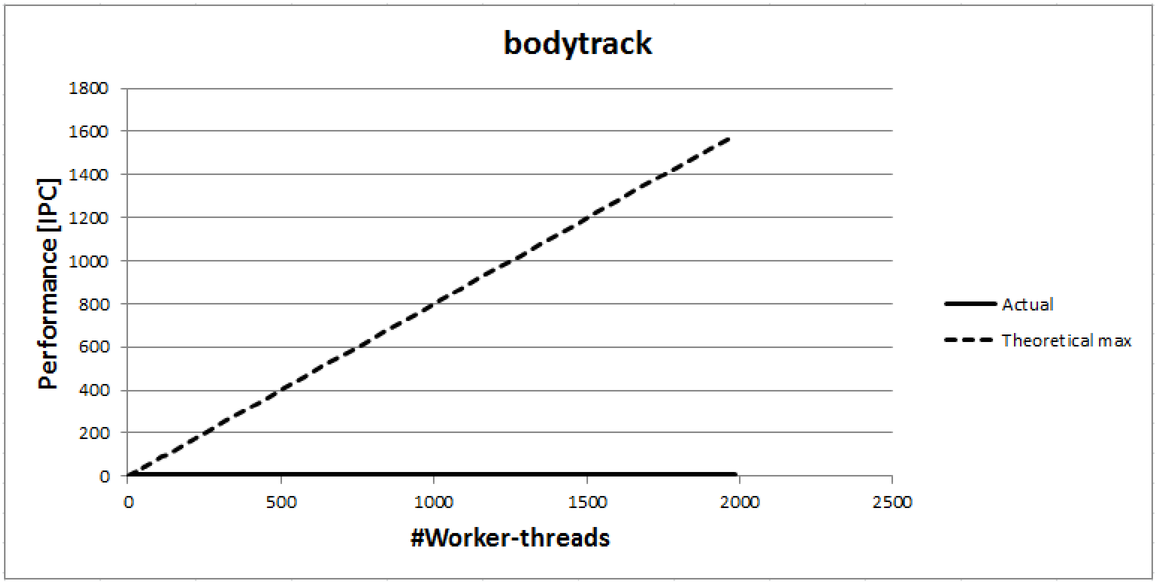


Figure 4-4: Parallelism scalability – bodytrack

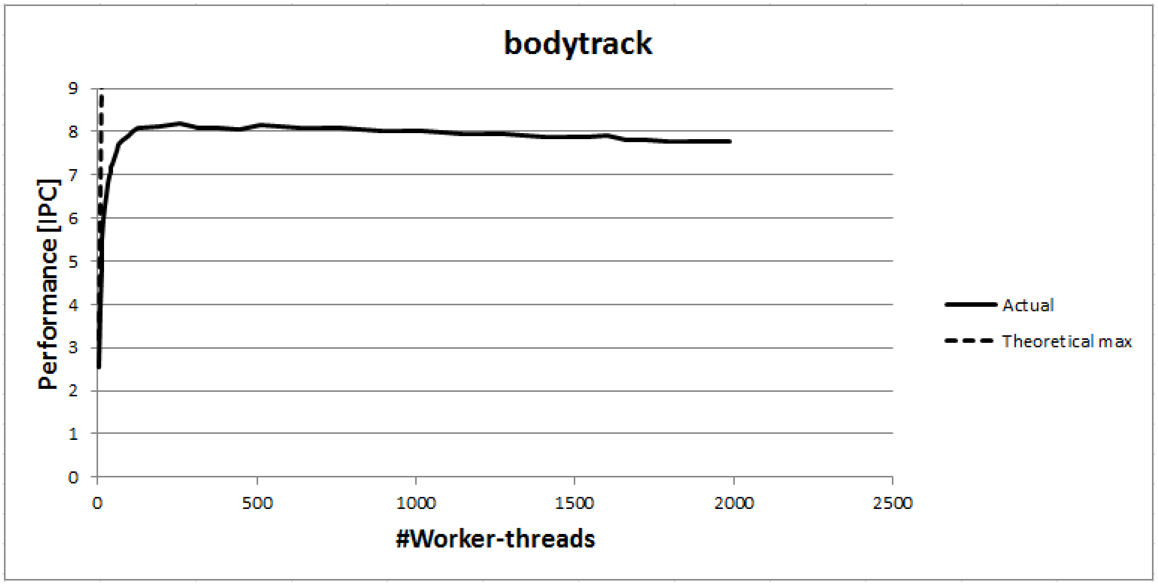
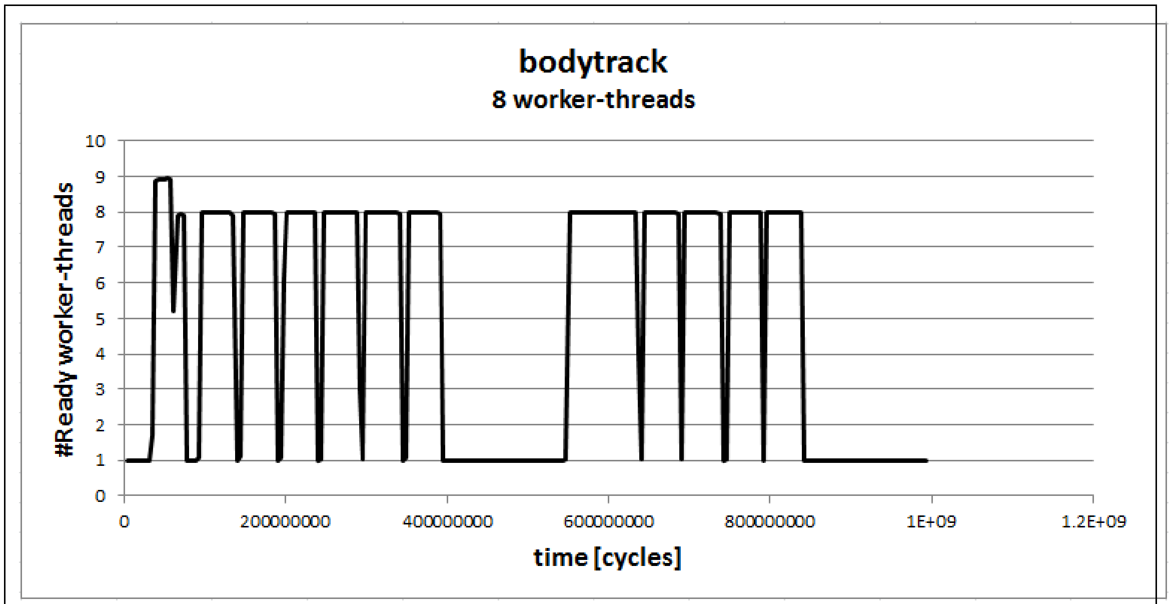


Figure 4-5: Parallelism scalability – bodytrack (scaled)

The **bodytrack** benchmark has poor scalability and the running threads over time curves in Figure 4-6 provides the explanation: there is a serial component that is ~$53 \cdot 10^7$ cycles long that increasingly dominates the execution time as the number of worker threads increases.


bodytrack
8 worker-threads
#Ready worker-threads
time [cycles]
0
1
2
3
4
5
6
7
8
9
10
0
200000000
400000000
600000000
800000000
1E+09
1.2E+09

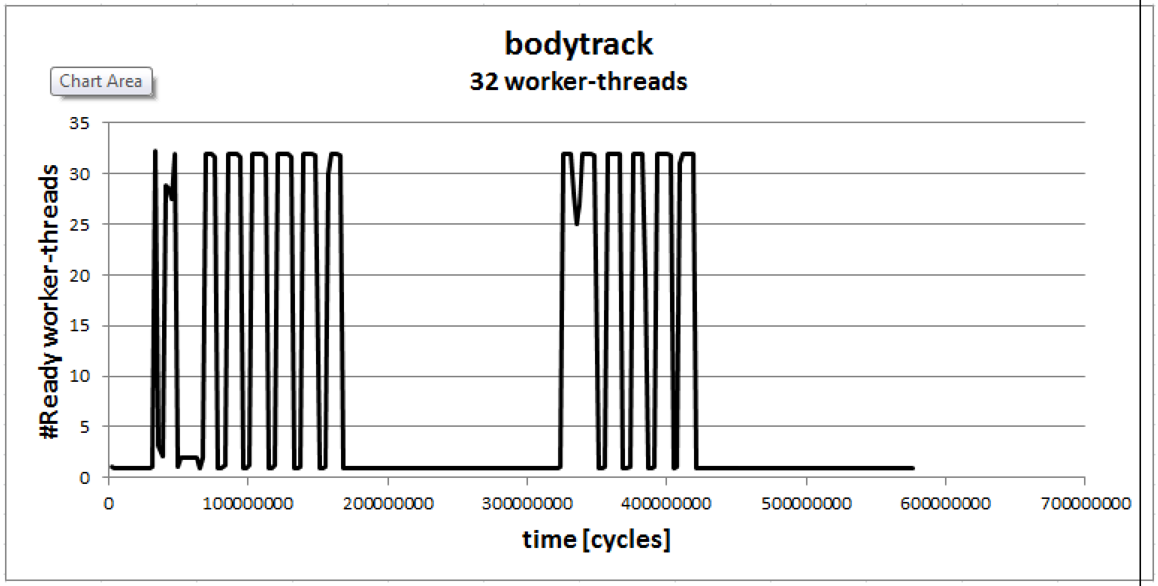


Chart Area
bodytrack
32 worker-threads
#Ready worker-threads
time [cycles]
0
5
10
15
20
25
30
35
0
100000000
200000000
300000000
400000000
500000000
600000000
700000000

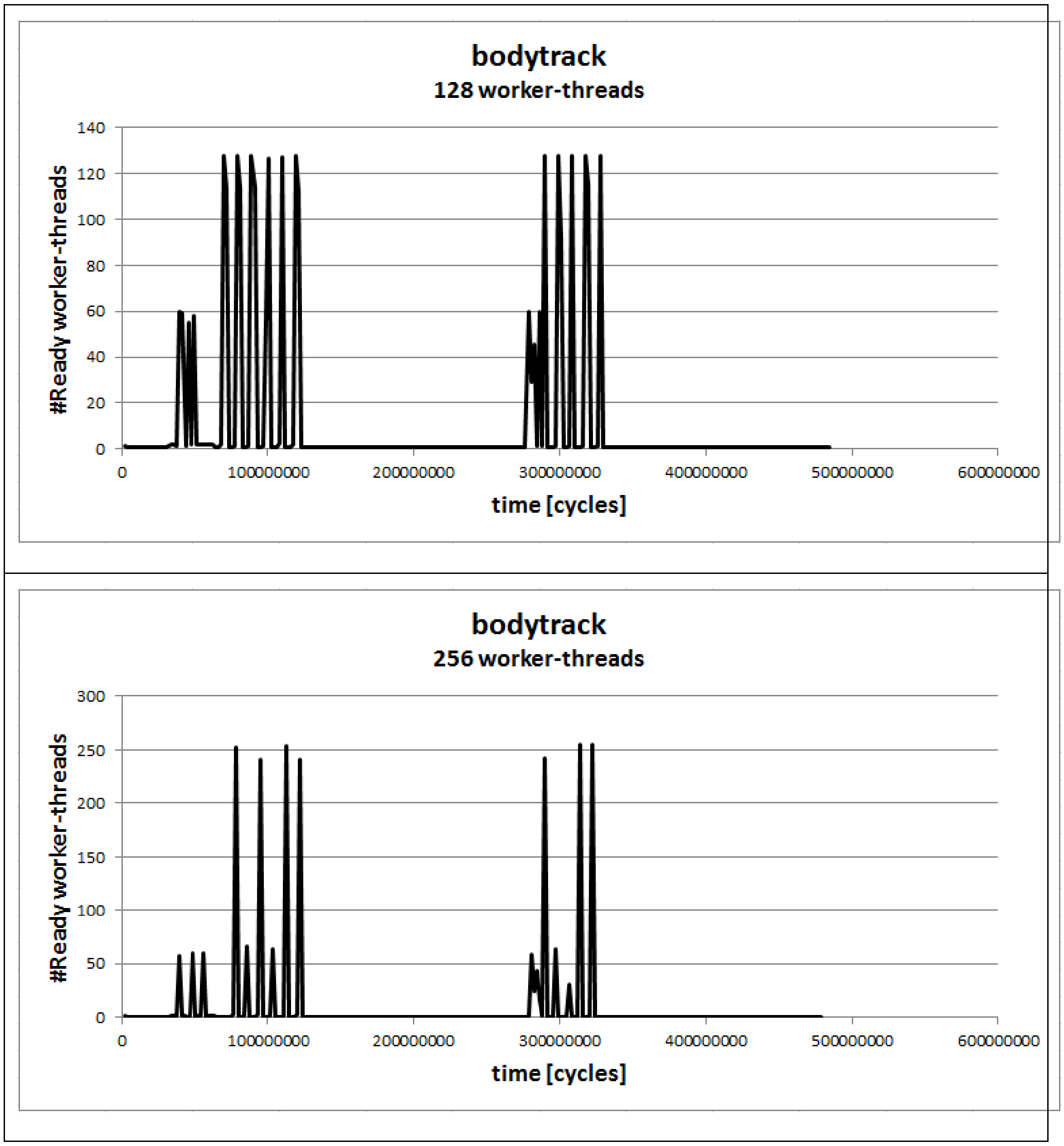


Figure 4-6: Running threads over time - bodytrack

4.3.3. canneal

The performance of the **canneal** benchmark is shown in Figure 4-7. Figure 4-8 is similar to Figure 4-7 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows that this benchmark has good scalability up to ~128 worker-threads and peak performance at ~256 threads. Beyond that the performance not only doesn't increase, it decreases.

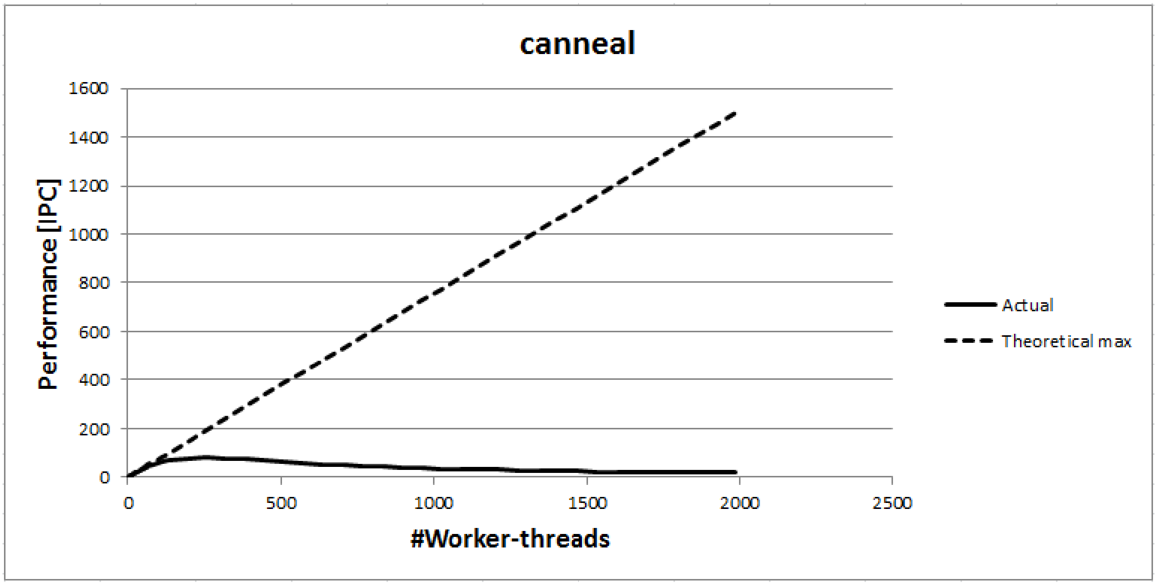


Figure 4-7: Parallelism scalability - canneal

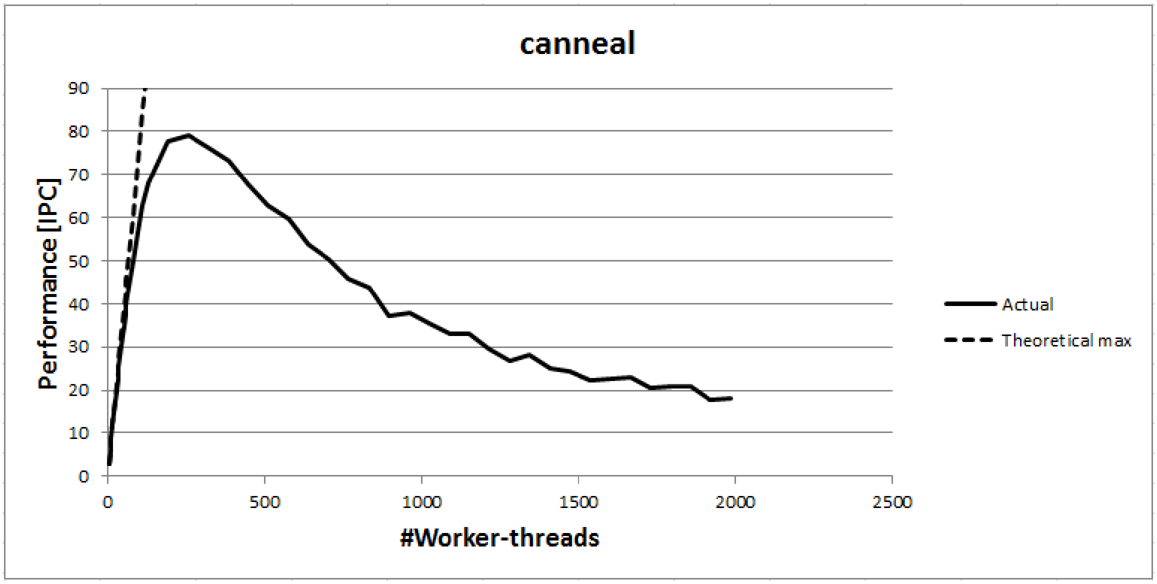
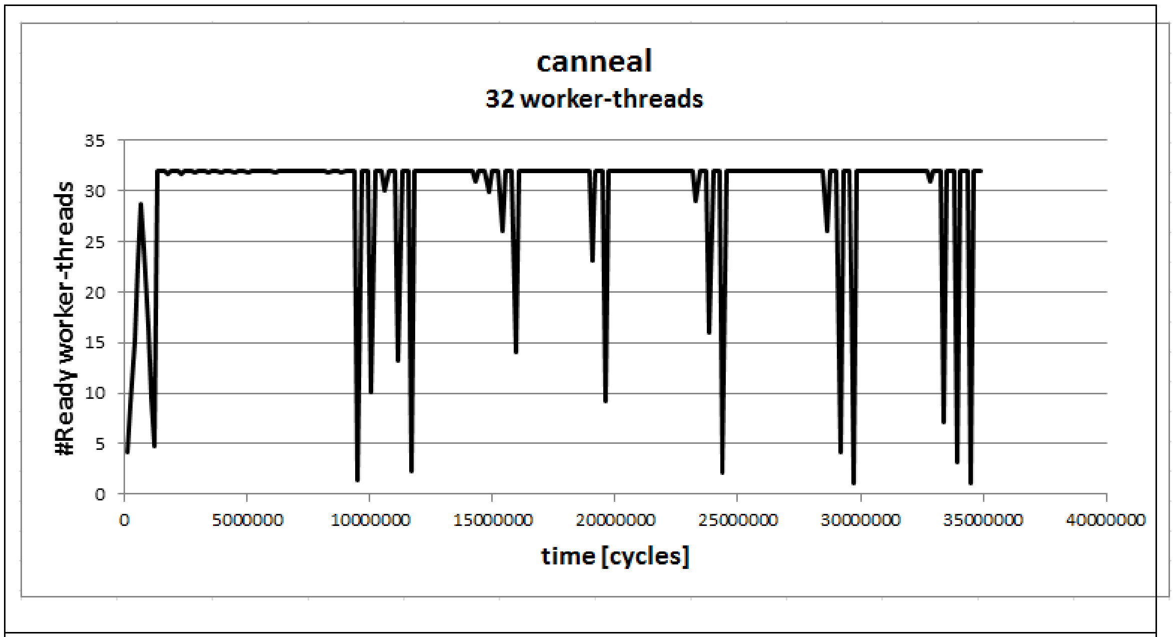
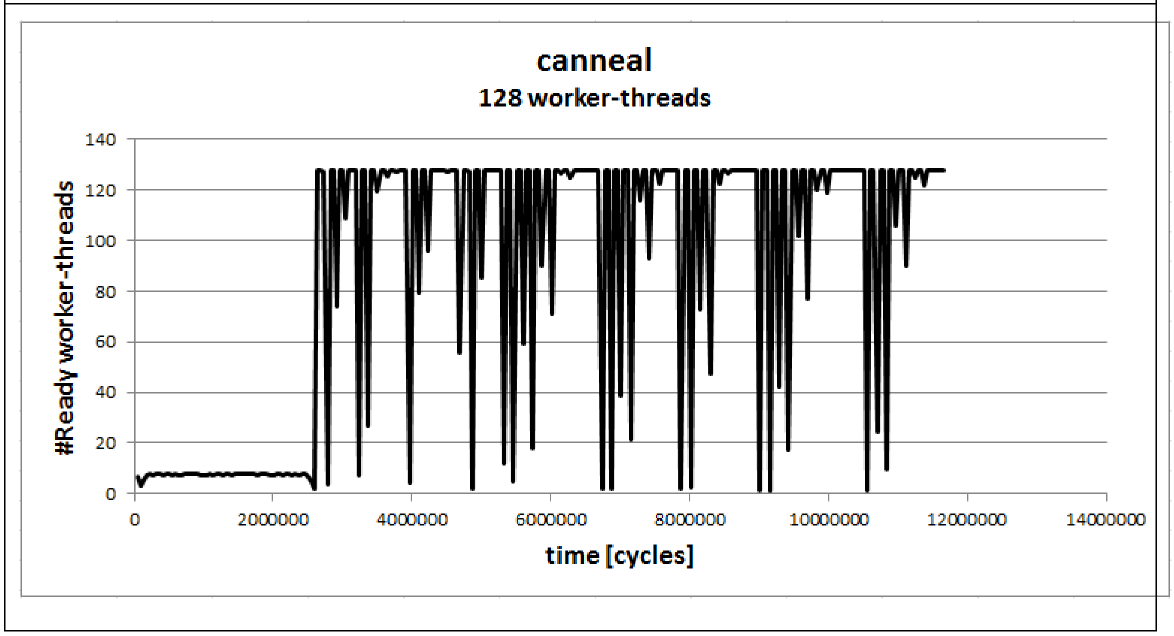
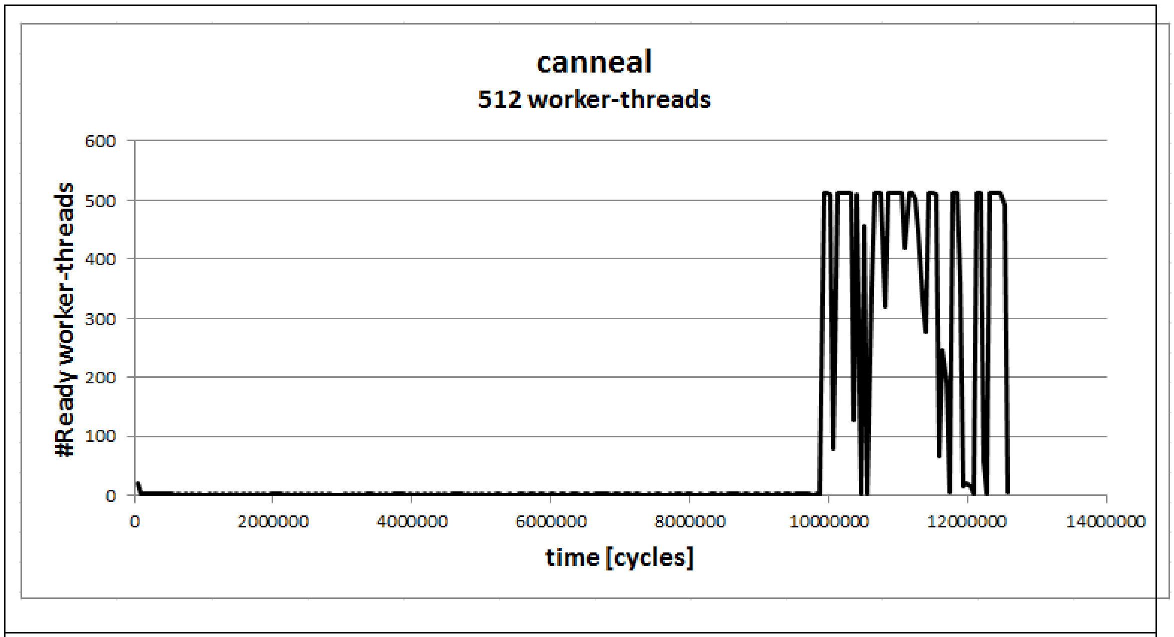


Figure 4-8: Parallelism scalability – canneal (scaled)

The running threads over time curve of Figure 4-9 explain the performance: this benchmark has a serial component that is not of fixed length but rather grows with the number of worker threads. It is completely masked with 32 worker threads but emerges and becomes increasingly dominant fast with the increase in worker-threads count.


canneal
32 worker-threads
#Ready worker-threads
time [cycles]
0
5
10
15
20
25
30
35
0
5000000
10000000
15000000
20000000
25000000
30000000
35000000
40000000


canneal
128 worker-threads
#Ready worker-threads
time [cycles]
0
20
40
60
80
100
120
140
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000



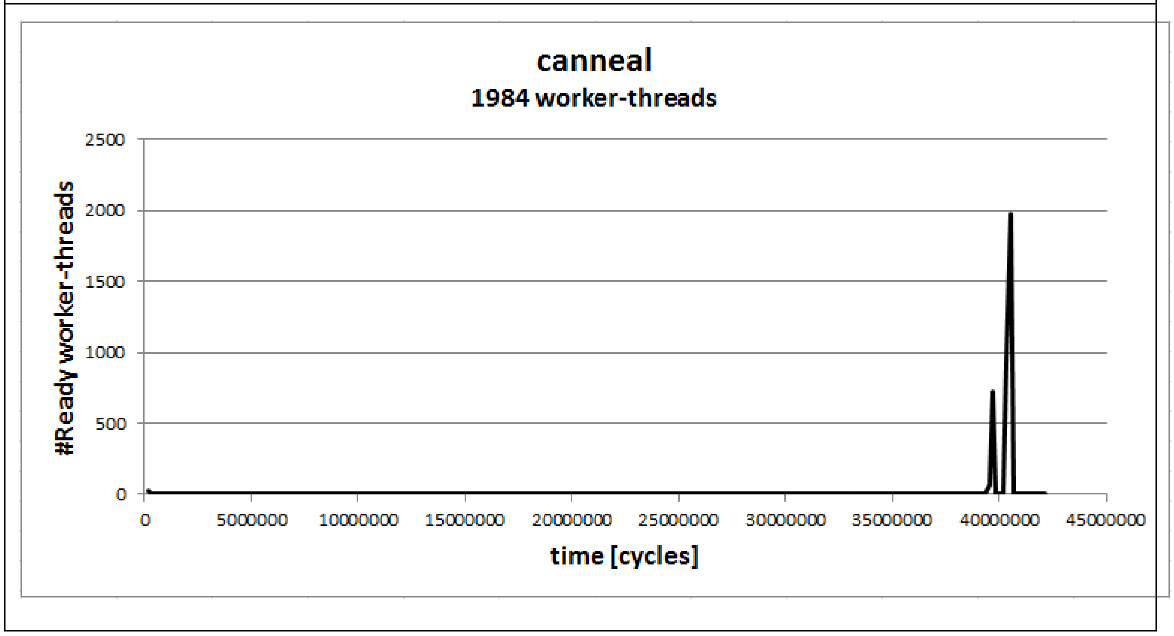


Figure 4-9: Running threads over time - canneal

4.3.4. dedup

The **dedup** benchmark has 3-stage pipelined design, each stage with as many worker threads as the parallelism degree specified in the benchmark invocation. Thus, this benchmark spawns three times the number of worker threads than the number specified in the program invocation. Since our simulator is limited to 2048 threads, this benchmark was simulated with up to 640 threads specified in the program invocation.

Due to the pipeline design, the end of ROI (Region-Of-Interest) was not taken to be when the first thread exits because there are several types of worker-threads (per pipeline stage) and some exit significantly earlier than others. Therefore, for this benchmark the special ROI inference based on worker thread exiting was not used. Instead the ROI end was inferred from the benchmark's built-in ROI end function call, i.e., after all worker-threads have exited.

The performance of the **dedup** benchmark is shown in Figure 4-10. Figure 4-11 is similar to Figure 4-10 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows that the performance plateaus at 64 threads.

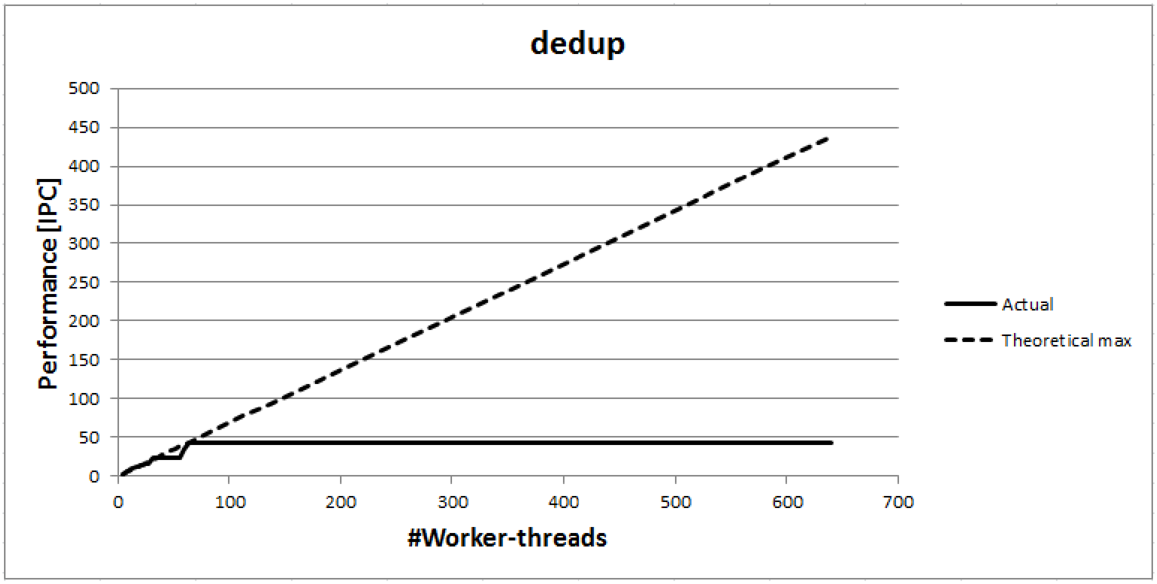


Figure 4-10: Parallelism scalability – dedup

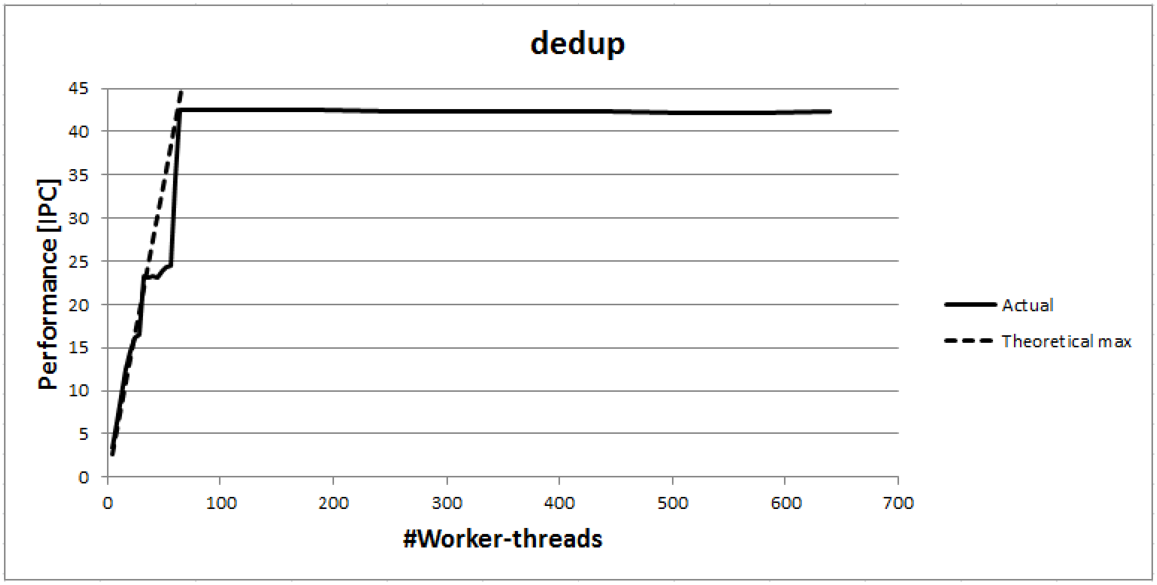
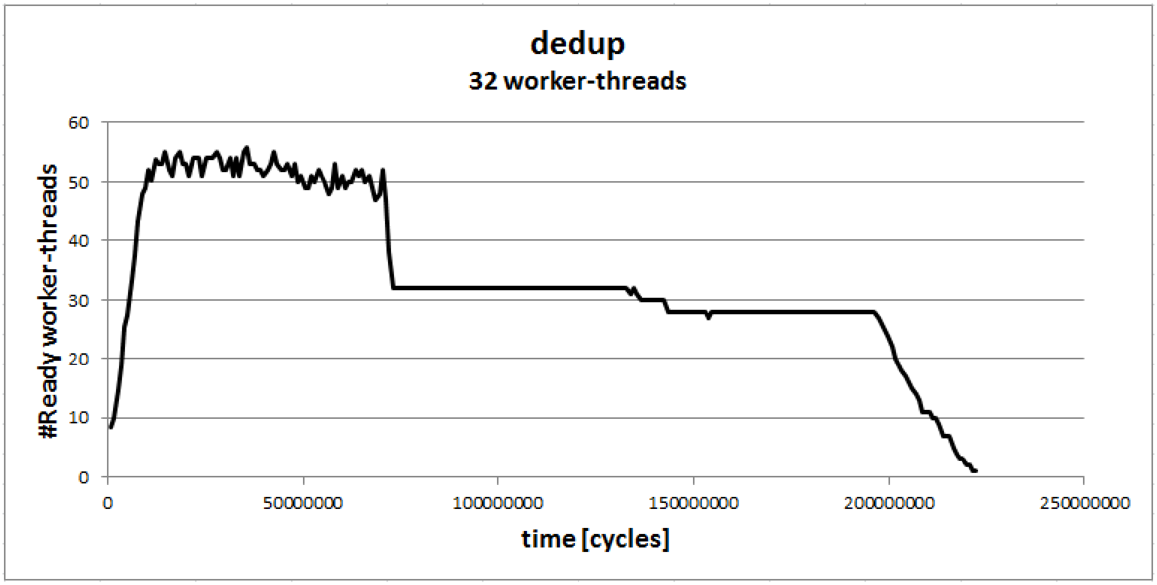
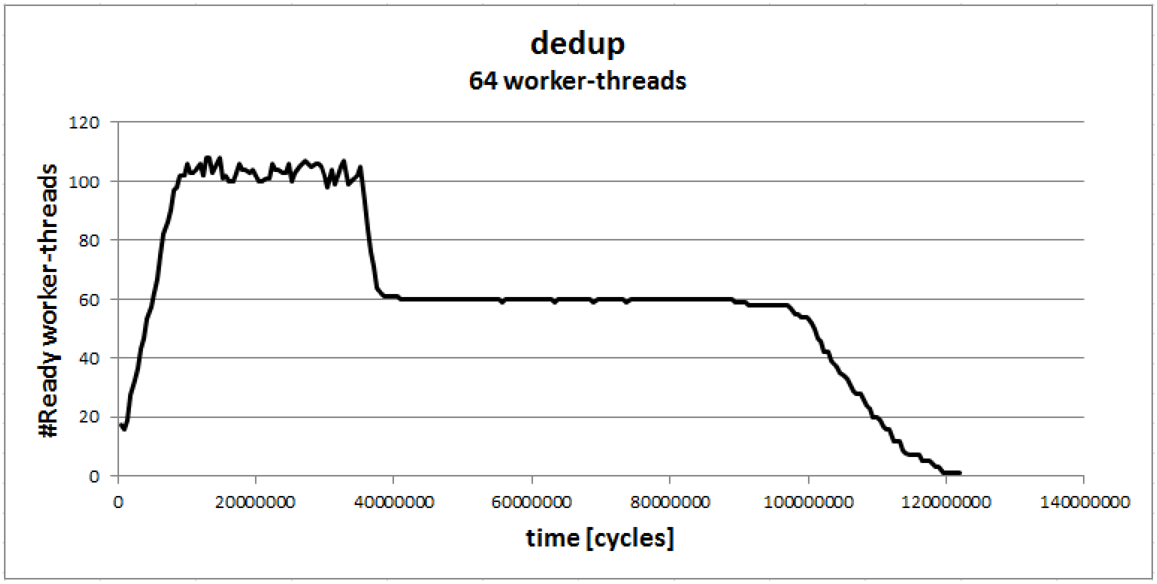


Figure 4-11: Parallelism scalability - dedup (scaled)

The running threads over-time curves are shown in Figure 4-12. Indeed they show that adding worker threads beyond 64 does not change the execution pattern. However, we see that there are periods of time with more running threads than the number of thread the benchmarks was invoked to because the benchmark actually spawns 3 times this number so although Figure 4-11 seems to indicate that up to 64 threads the performance scalability is very good, in fact it requires more cores to reach that performance.


dedup
32 worker-threads
#Ready worker-threads
60
50
40
30
20
10
0
0
50000000
100000000
150000000
200000000
250000000
time [cycles]


dedup
64 worker-threads
#Ready worker-threads
120
100
80
60
40
20
0
0
20000000
40000000
60000000
80000000
100000000
120000000
140000000
time [cycles]

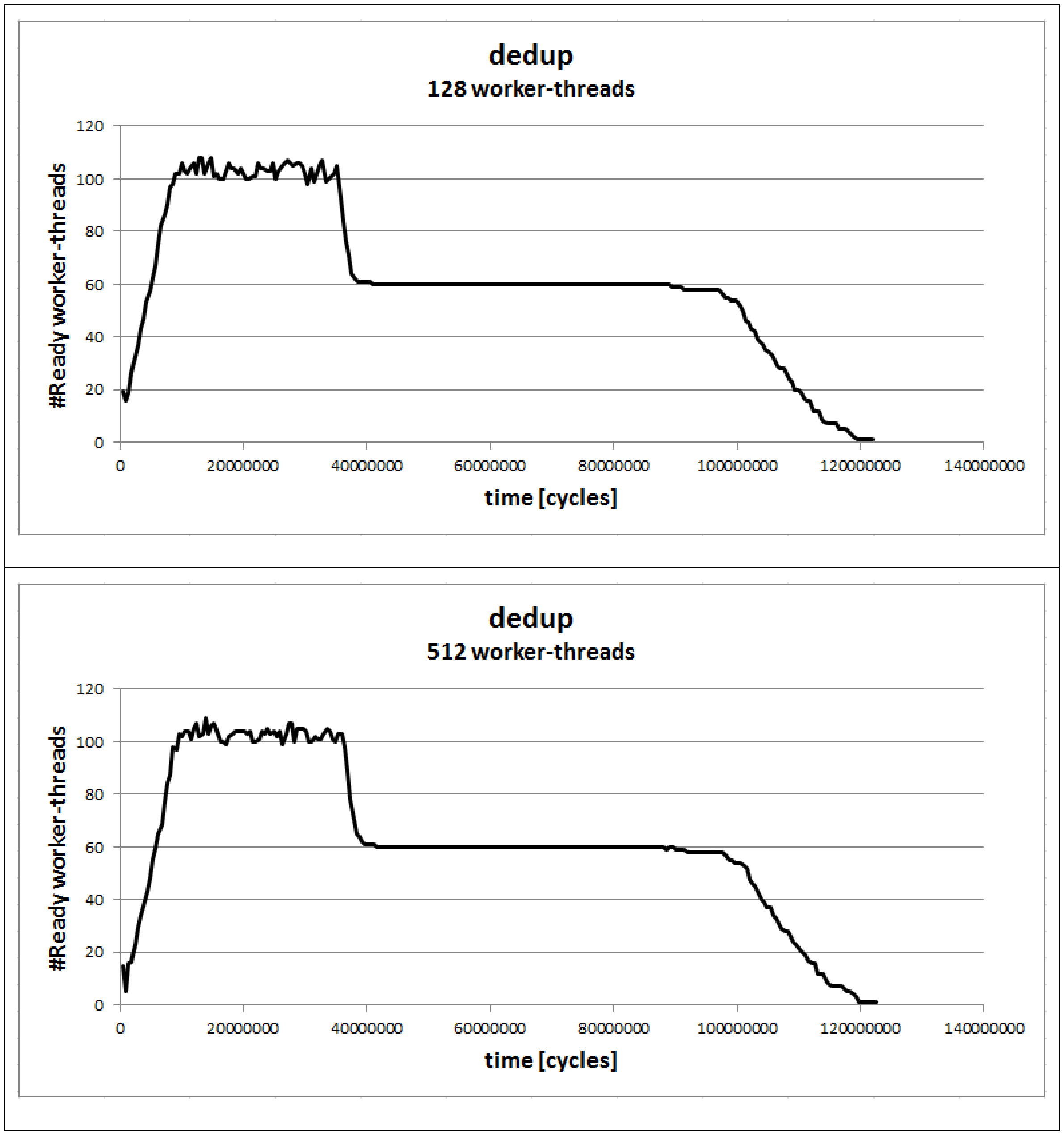


Figure 4-12: Running threads over time - dedup

4.3.5. facesim

The **facesim** benchmark requires the number of worker threads to be a power of 2 and is limited to 128 worker-threads. This limit is both hard-coded and there is no suitable data-set for more than 128 worker-threads. Thus, merely changing the hard-coded limit and invoking with more threads fails to run due to lack of appropriate data set.

The performance of the **facesim** benchmark is shown in Figure 4-13. Figure 4-14 is similar to Figure 4-13 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows very poor performance scalability – while the performance is monotonously increasing, the marginal performance increase decreases rapidly and the maximal performance, achieved at 128 threads is equal to the theoretical maximal performance at ~8 threads.

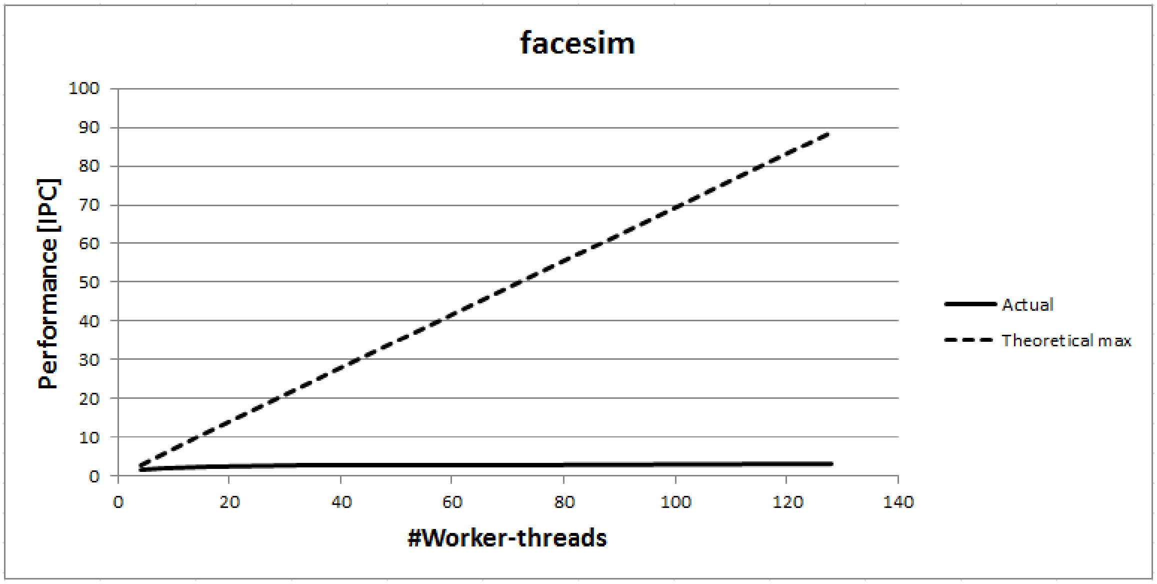


Figure 4-13: Parallelism scalability – facesim

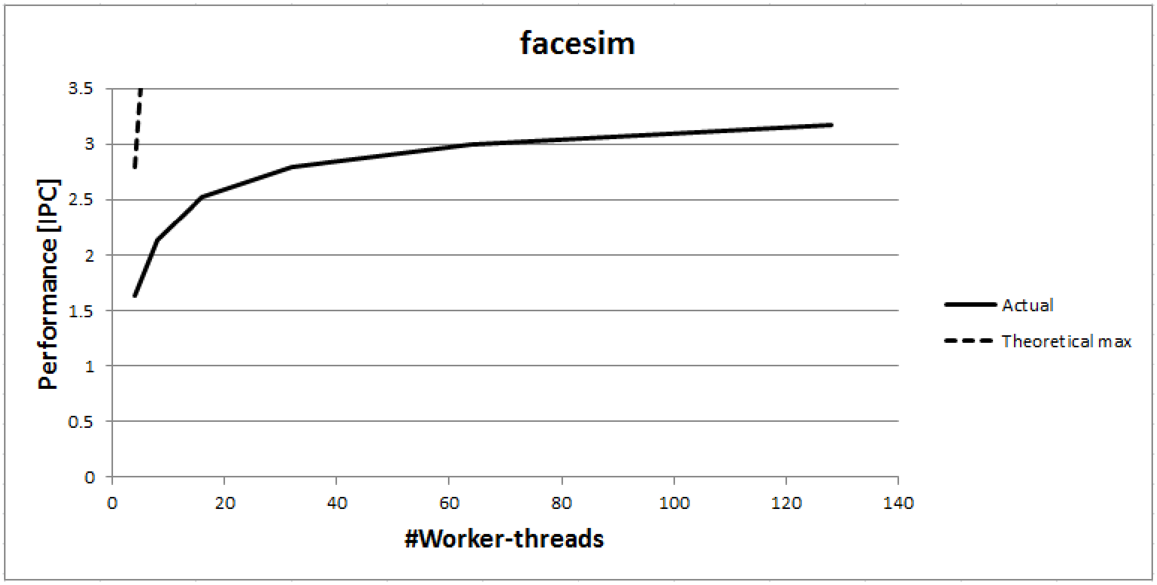
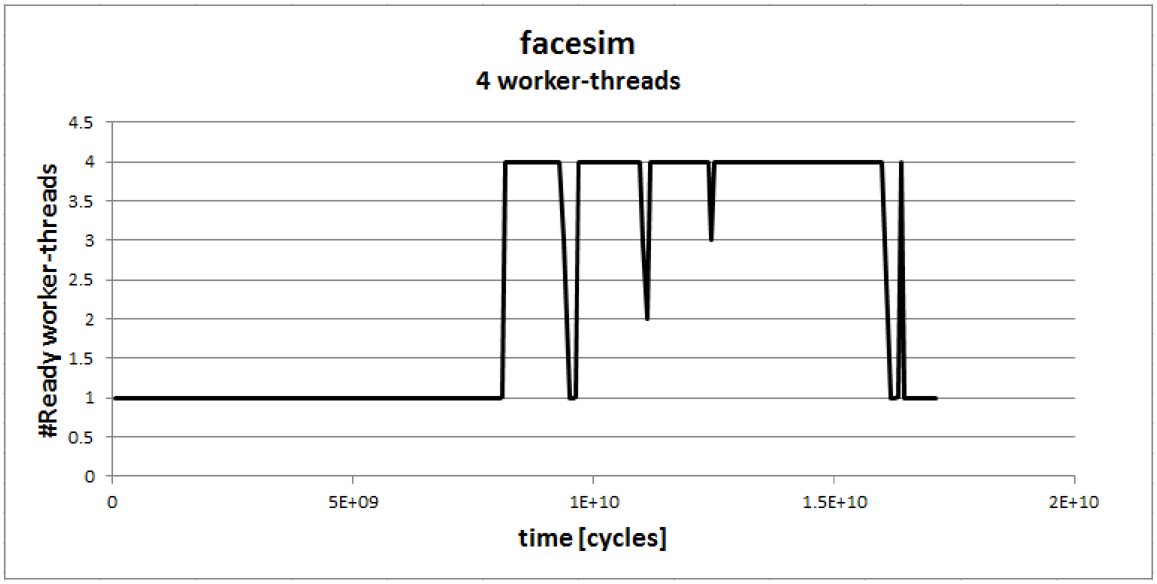
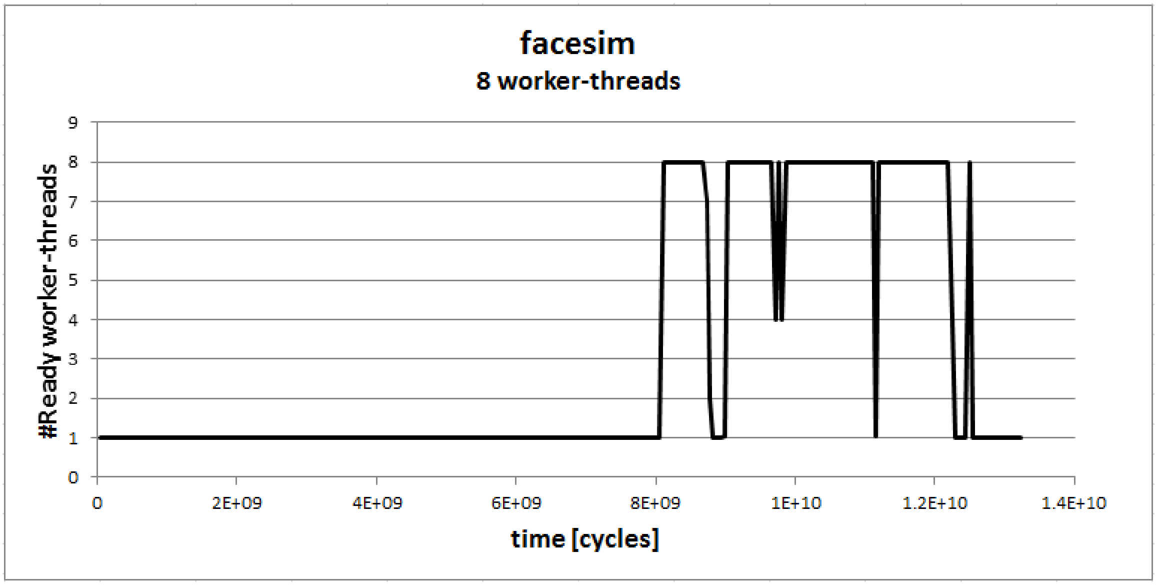
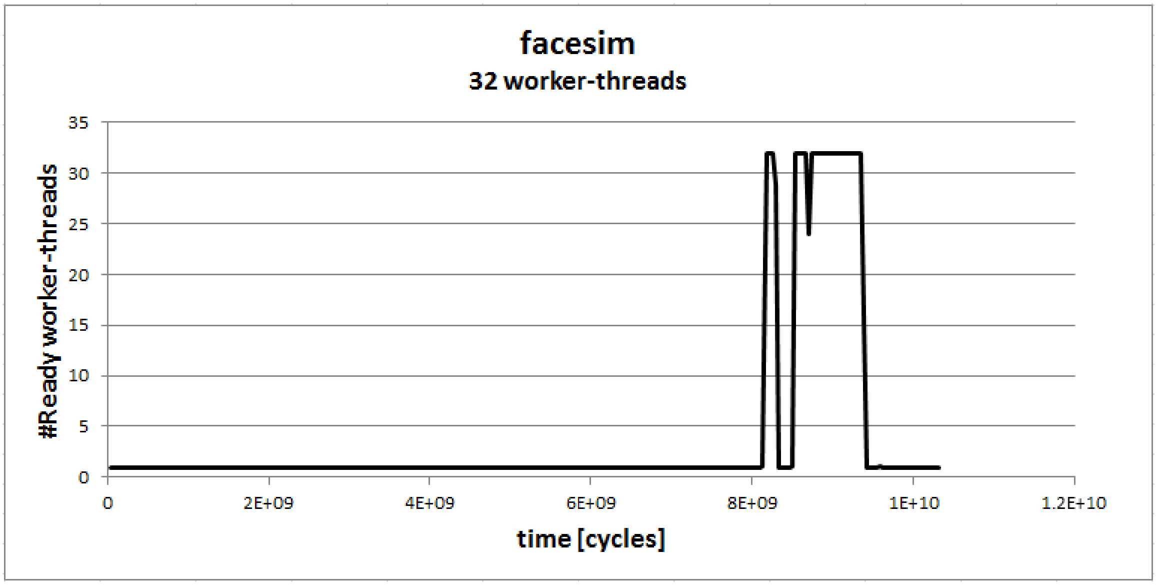


Figure 4-14: Parallelism scalability - facesim (scaled)

The running threads over-time curves are shown in Figure 4-15. They show that there is a leading serial part of ~11E+10 cycles long that occupies around half of the execution time with 4 worker threads and increasingly dominates the execution time, which explains the poor performance scalability.







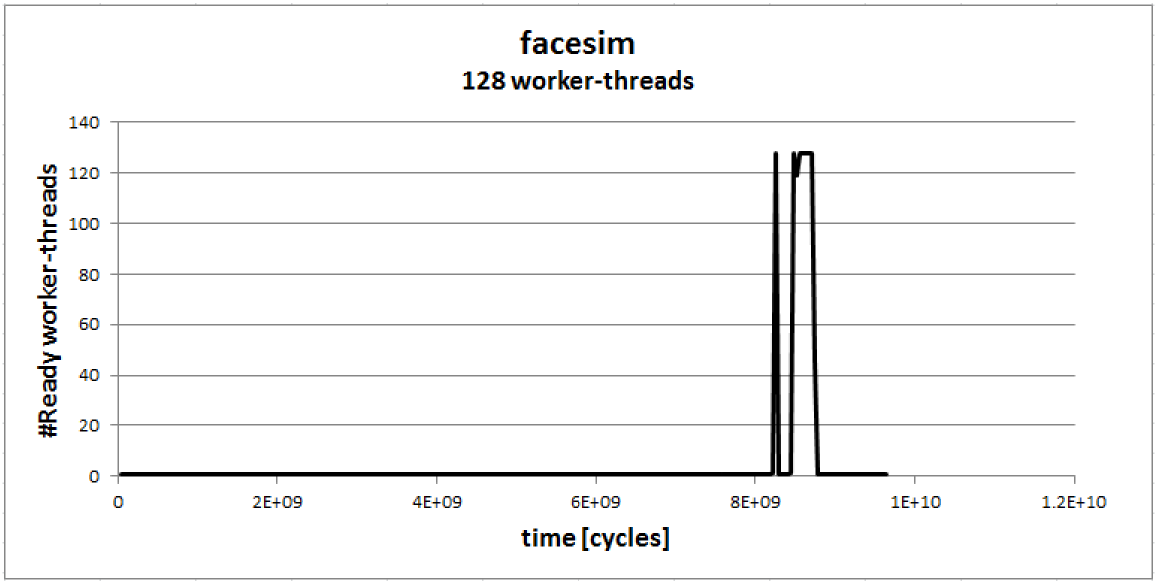


Figure 4-15: Running threads over time - facesim

4.3.6. ferret

The **ferret** benchmark has 4-stage pipelined design, each stage with as many worker threads as the parallelism degree specified in the benchmark invocation. Thus, this benchmark spawns four times the number of worker threads than the number specified in the program invocation. Since our simulator is limited to 2048 threads, this benchmark was simulated with up to 448 threads specified in the program invocation.

The performance of the **ferret** benchmark is shown in Figure 4-16. Figure 4-17 is similar to Figure 4-16 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows that the performance plateaus at 8 threads and there is very little performance difference between 4 and 8 threads. Thus, this benchmark has poor performance scalability.

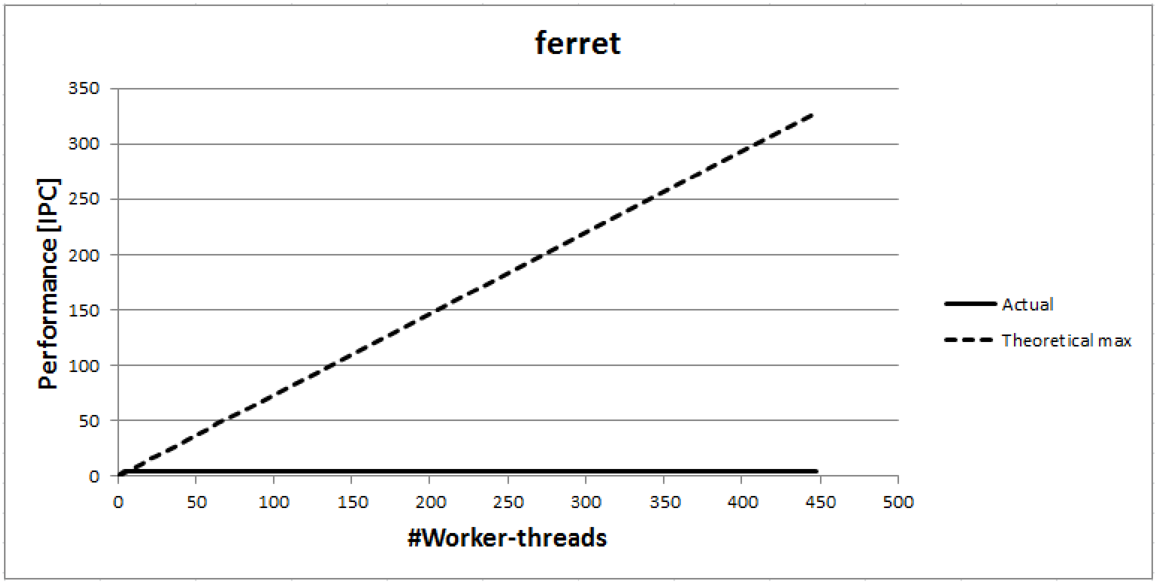


Figure 4-16: Parallelism scalability - ferret

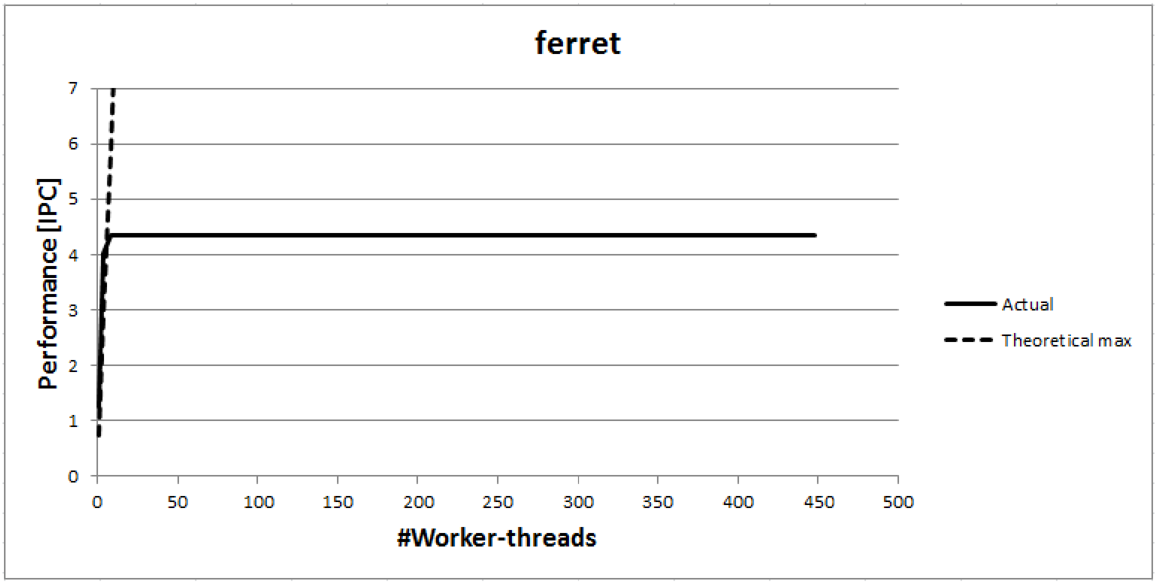
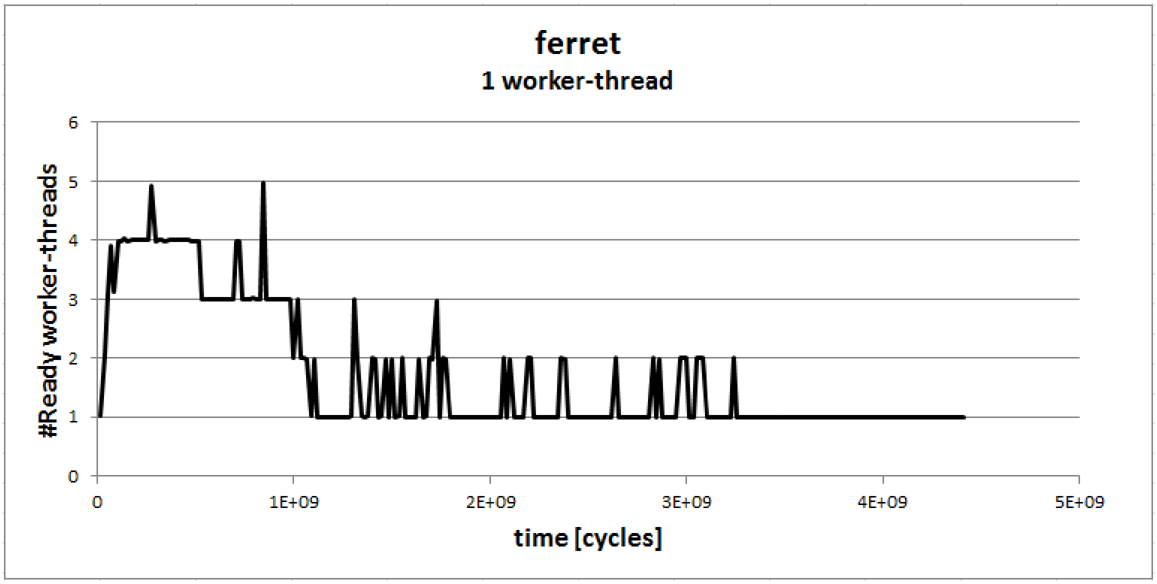
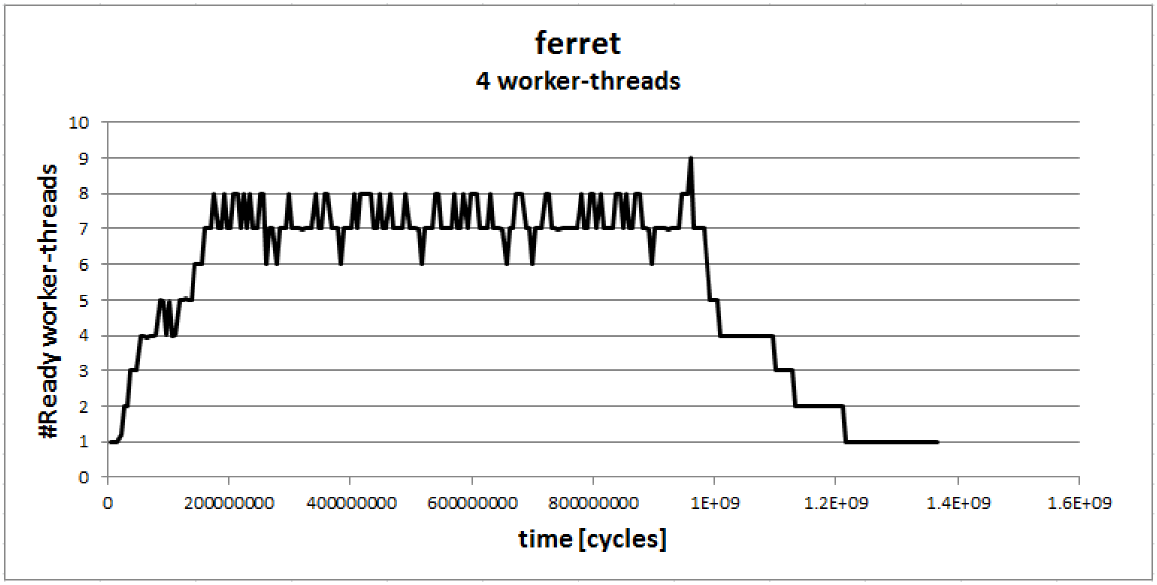
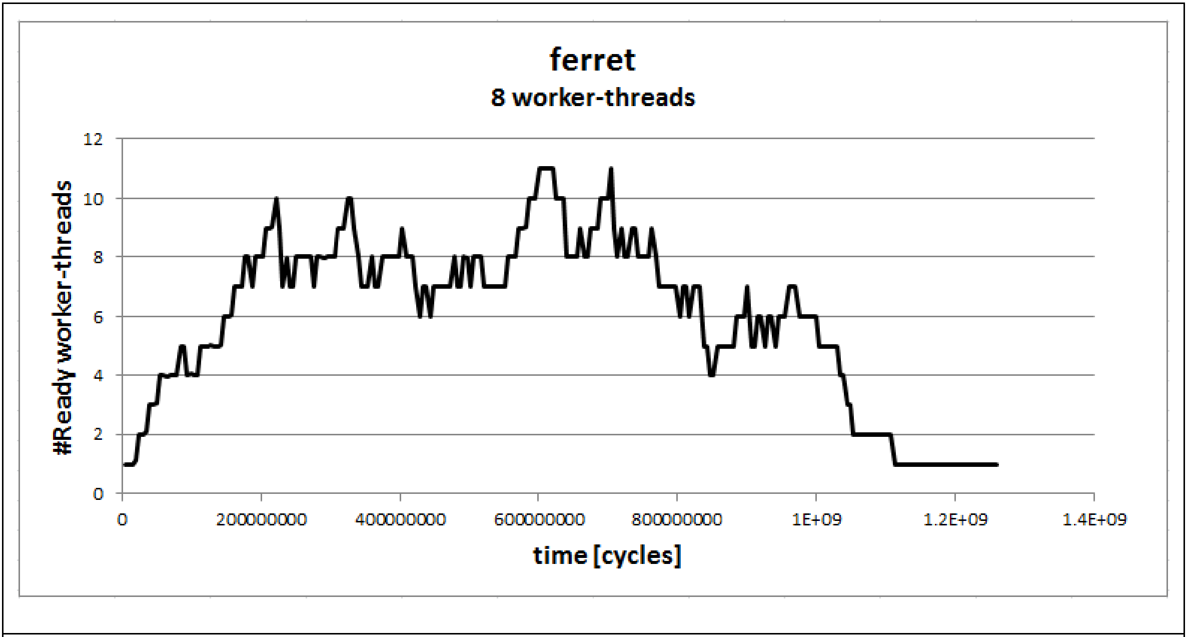


Figure 4-17: Parallelism scalability – ferret (scaled)

The running threads over-time curves are shown in Figure 4-18. Indeed they show that most of the time there are no more than 8 threads running and at most 12 are running at any given time. Also, they show the same execution pattern for 8 threads as for 256 threads, which Figure 4-17 shows that both result the same performance.







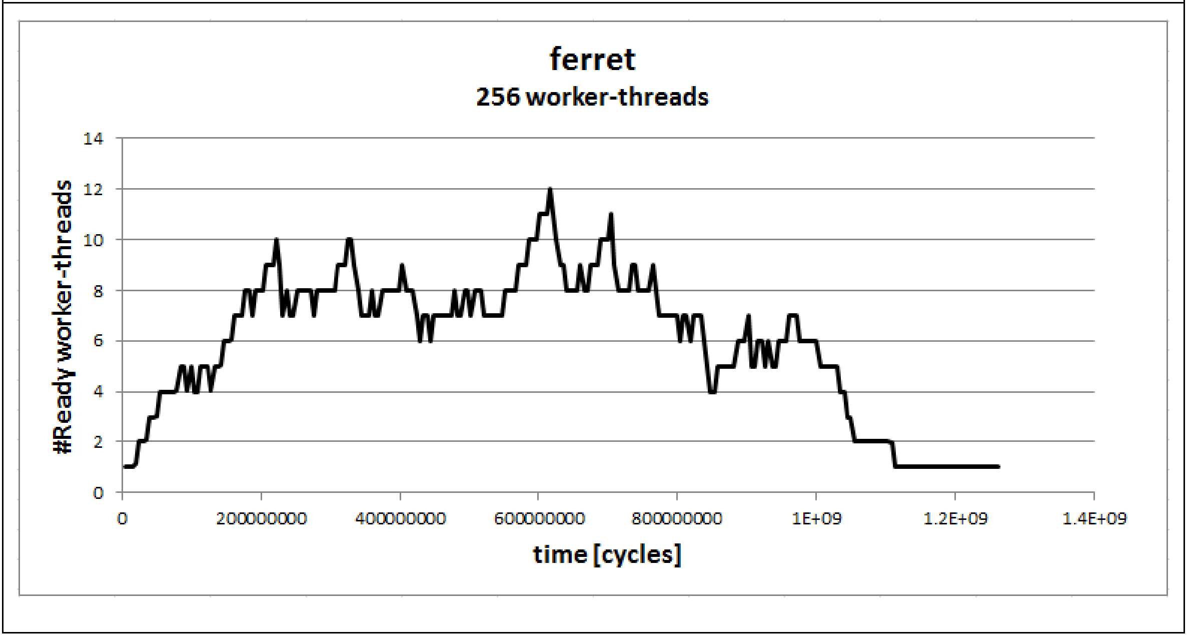


Figure 4-18: Running threads over time - ferret

4.3.7. fluidanimate

The **fluidanimate** benchmark requires the number of worker threads to be a power of 2. Also, it doesn't support 2048 threads: it requires an *image block* per worker-thread and in the *simmedium* and *simlarge* data sets there are not enough image blocks for 2048 worker-threads. Therefore, this benchmark is simulated with up to 1024 threads.

The performance of the **fluidanimate** benchmark is shown in Figure 4-19. It shows very good performance scalability – almost linear and close to the theoretical maximum performance.

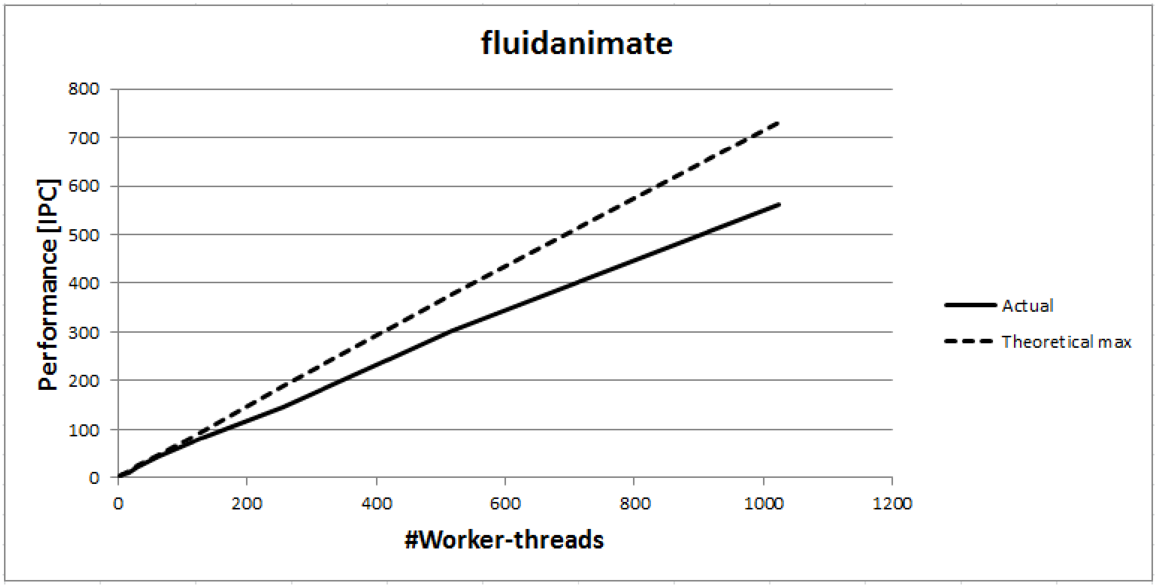
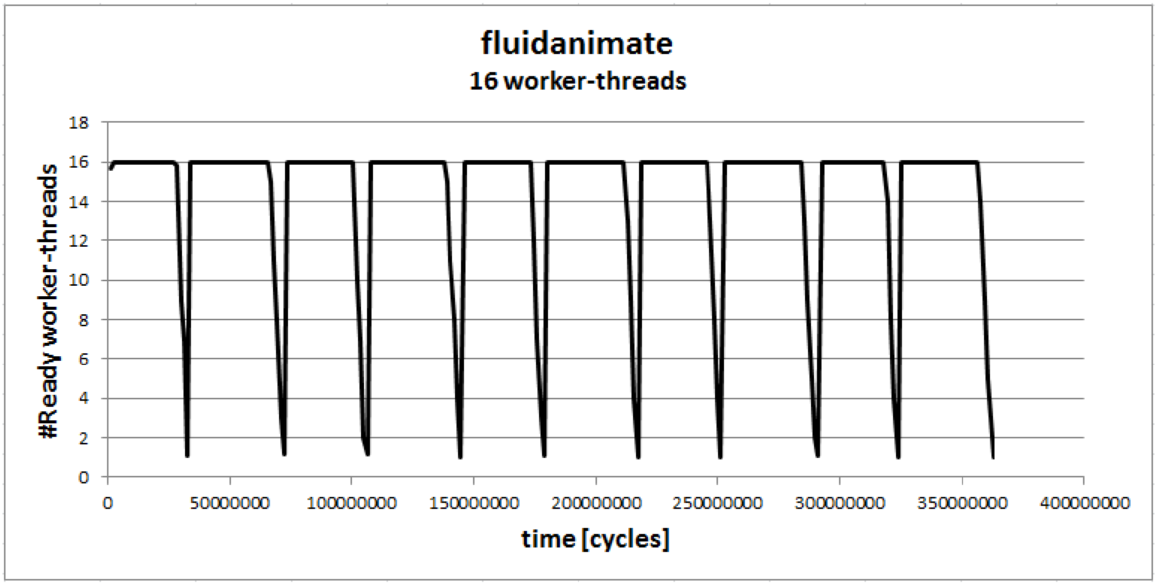
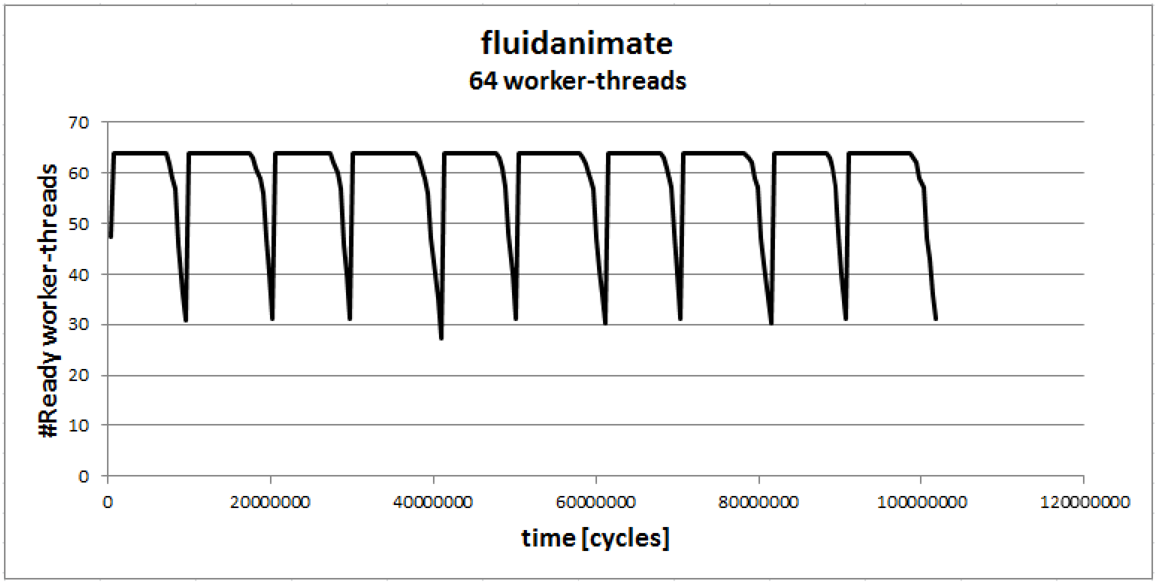
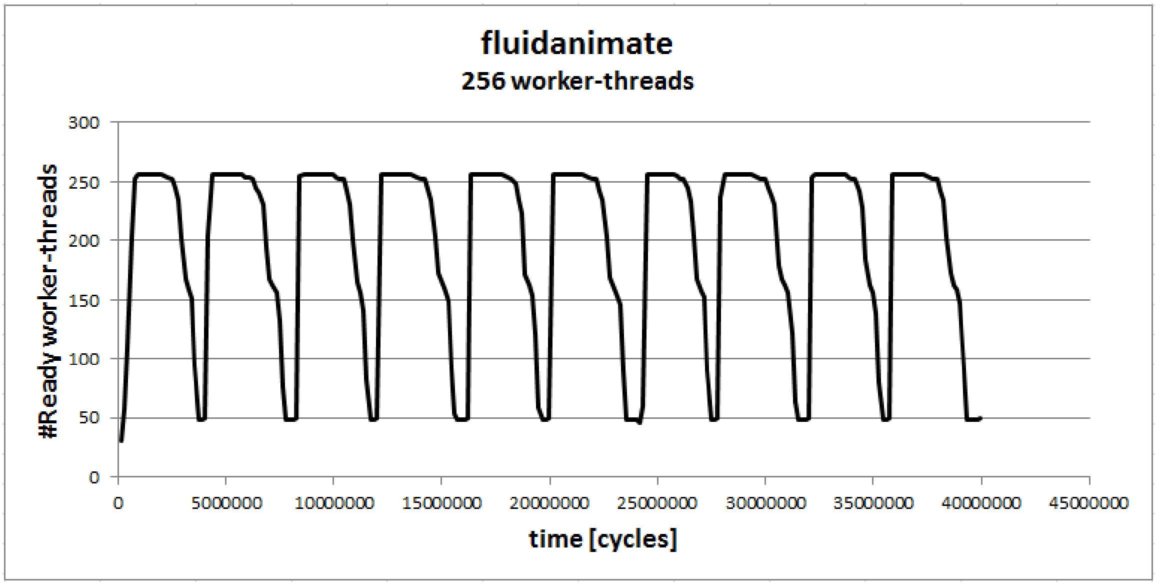


Figure 4-19: Parallelism scalability - fluidanimate

The running threads over-time curves are shown in Figure 4-20. They show that there is no serial phase, and most of the time all threads are running, which explains the good scalability. The fraction of the time that all threads are running decreases, which explains the increasing gap between the theoretical maximum and actual performance.


fluidanimate
16 worker-threads
#Ready worker-threads
time [cycles]
0
2
4
6
8
10
12
14
16
18
0
50000000
100000000
150000000
200000000
250000000
300000000
350000000
400000000


fluidanimate
64 worker-threads
#Ready worker-threads
time [cycles]
0
10
20
30
40
50
60
70
0
20000000
40000000
60000000
80000000
100000000
120000000



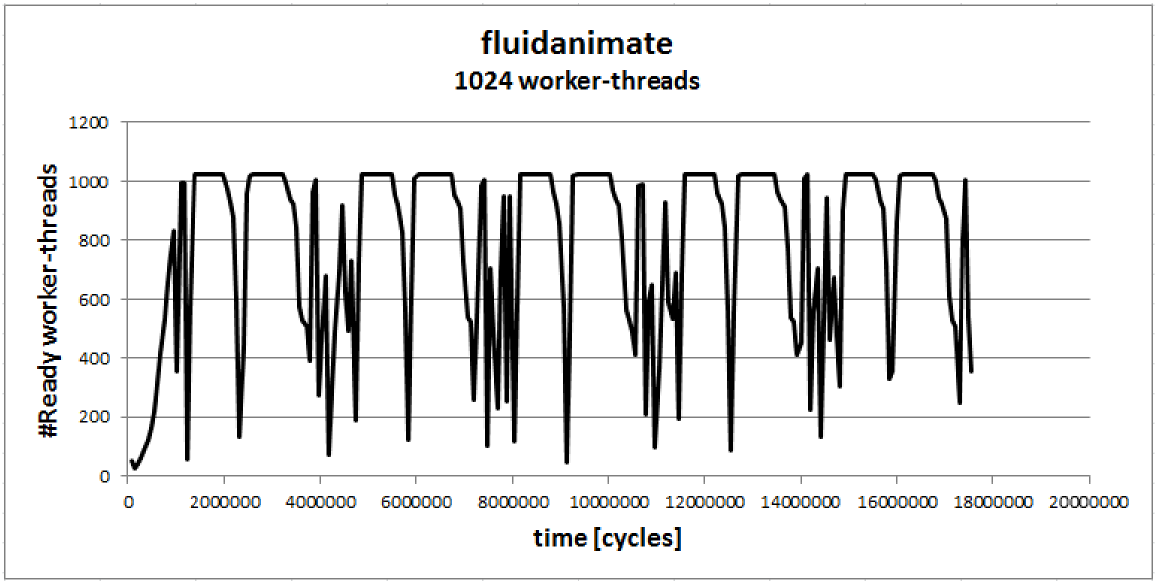


Figure 4-20: Running threads over time - fluidanimate

4.3.8. raytrace

The **raytrace** benchmark contains a barrier at the beginning of the worker-threads in order to maximize parallelism by making all the threads start only after all have been created and reached the starting point. As described in section 4.2, a barrier in the workload itself introduces a serial phase so for our measurements we removed the barrier from the benchmark and replaced it by the simulator's support for worker-threads start synchronization.

The performance of the **raytrace** benchmark is shown in Figure 4-21. Figure 4-22 is similar to Figure 4-21 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows good performance scalability up to ~80 threads at which point it virtually plateaus.

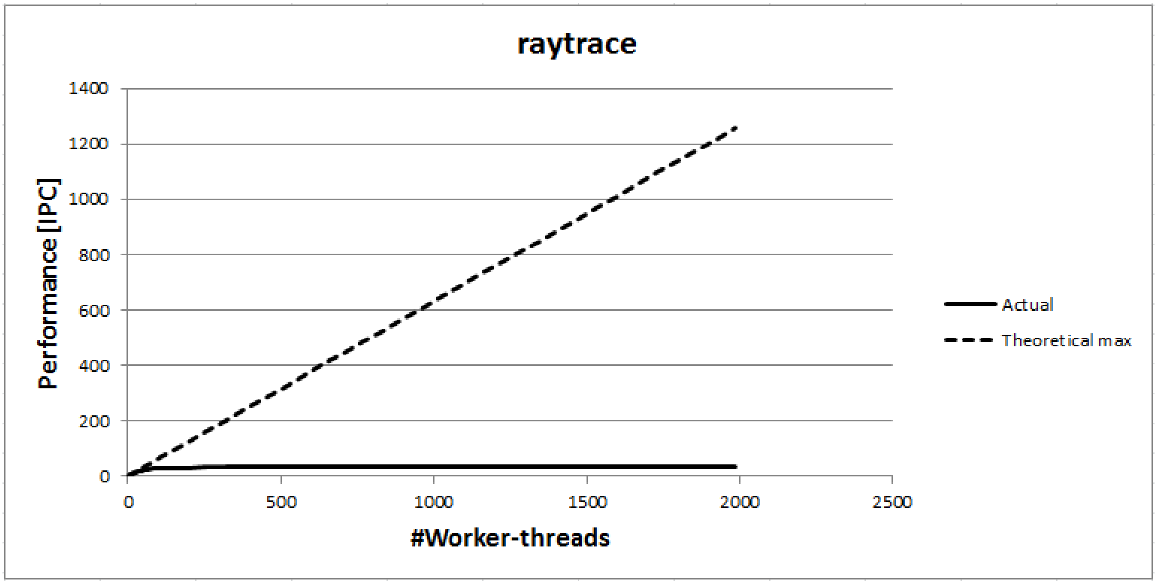


Figure 4-21: Parallelism scalability - raytrace

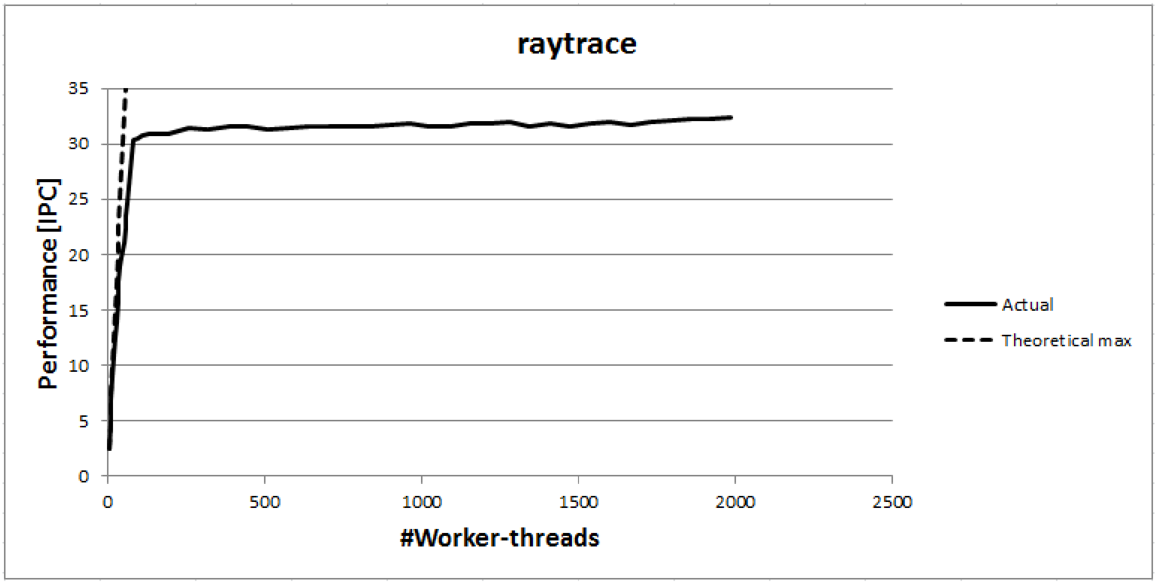
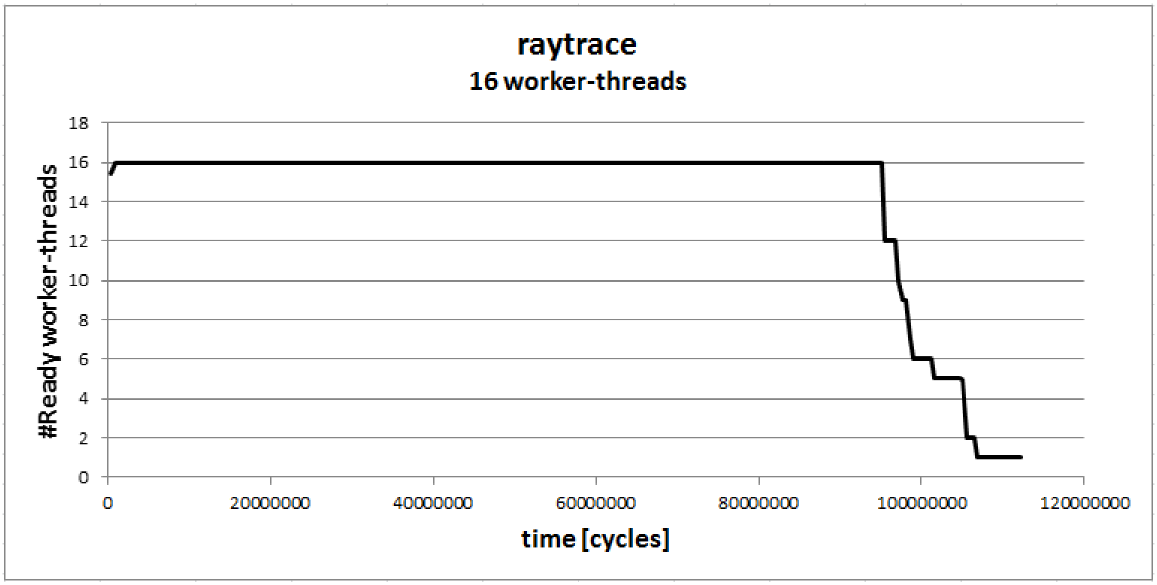
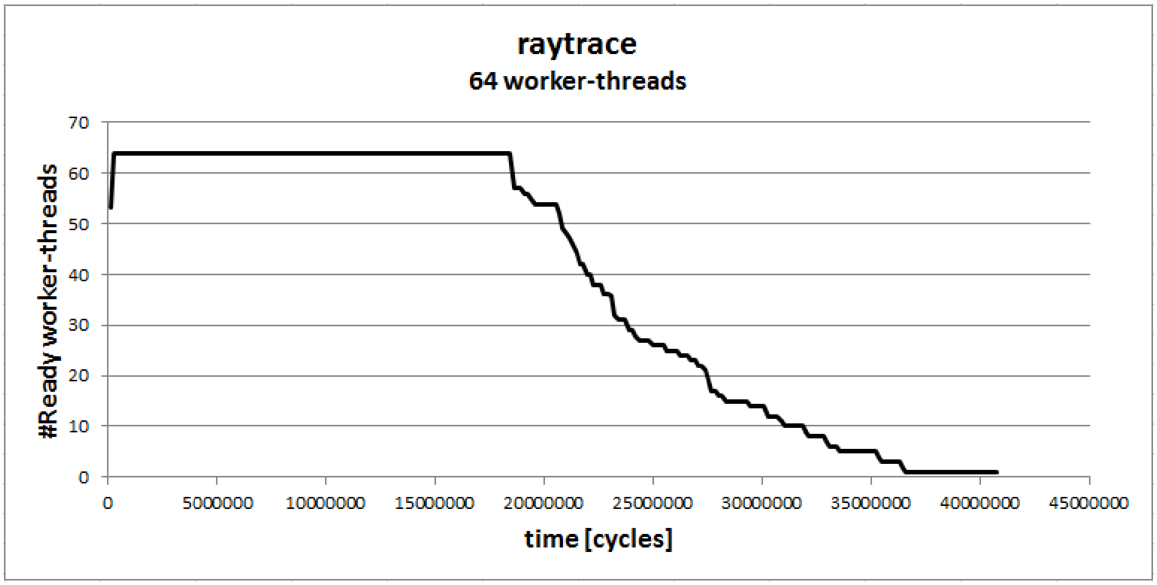
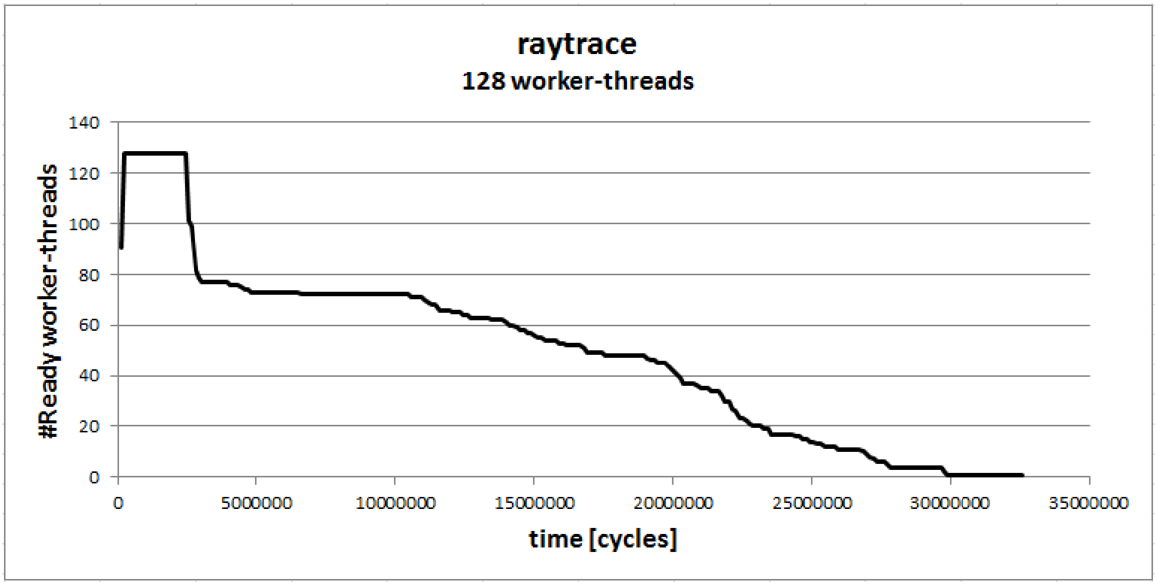


Figure 4-22: Parallelism scalability – raytrace (scaled)

The running threads over-time curves are shown in Figure 4-23. They show that as the number of worker-threads increases, there is a tail that above 128 threads is not affected by the number of threads and increasingly dominant the execution time.







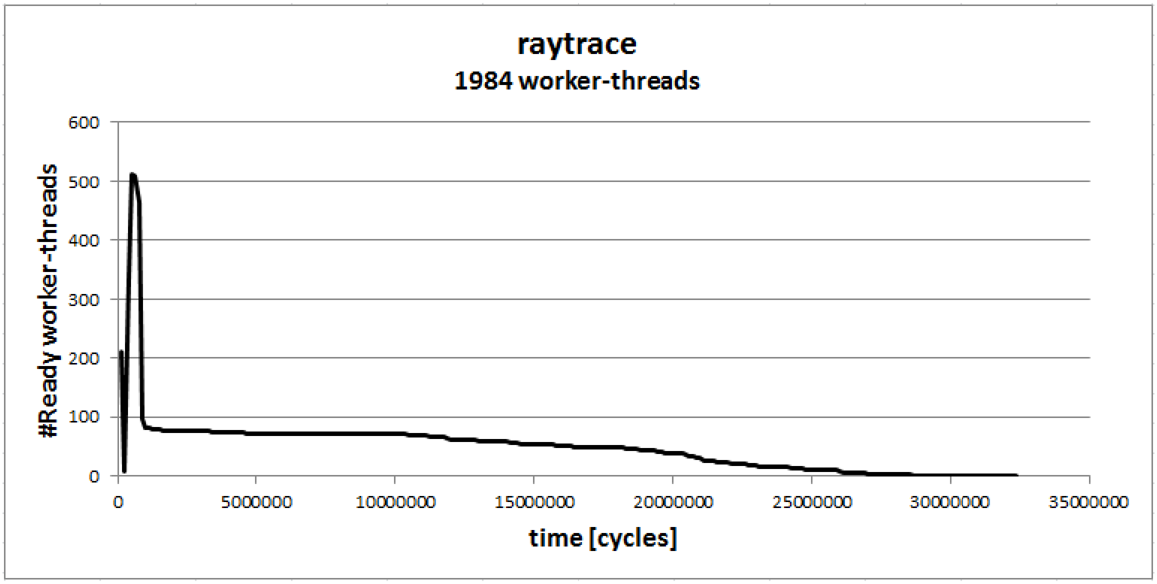


Figure 4-23: Running threads over time - raytrace

4.3.9. streamcluster

The **streamcluster** benchmark has a data-set that is proportional to the number of worker-threads and therefore its execution time is proportional to the number of threads. Large number of threads takes exceedingly long time to simulate. Therefore, this benchmark is simulated with up to 640 threads.

The performance of the **streamcluster** benchmark is shown in Figure 4-24. It shows good performance scalability – linear and about half of the maximum performance.

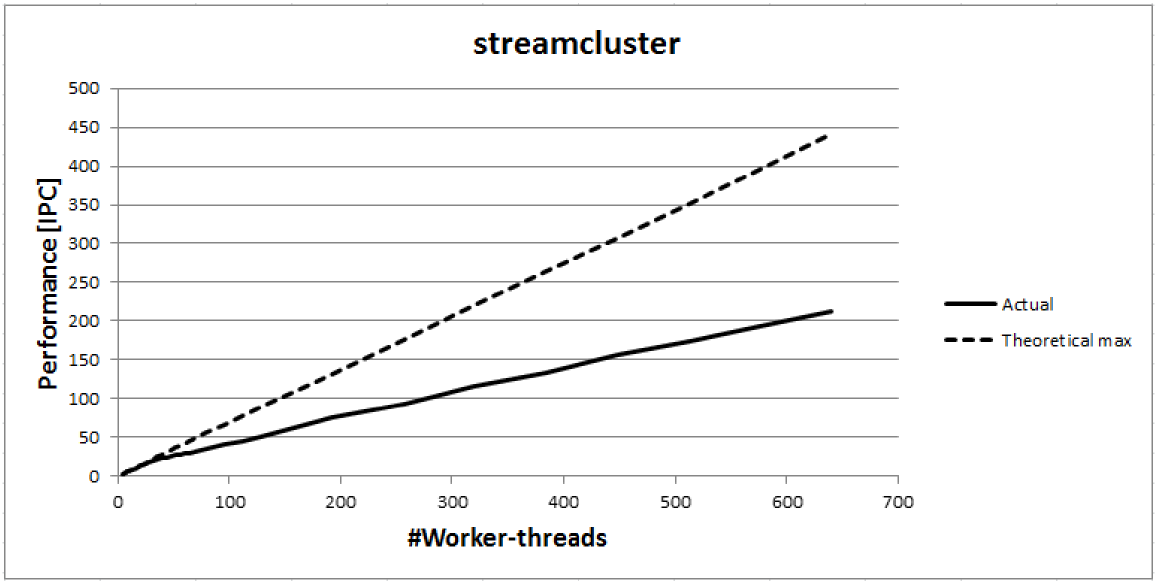
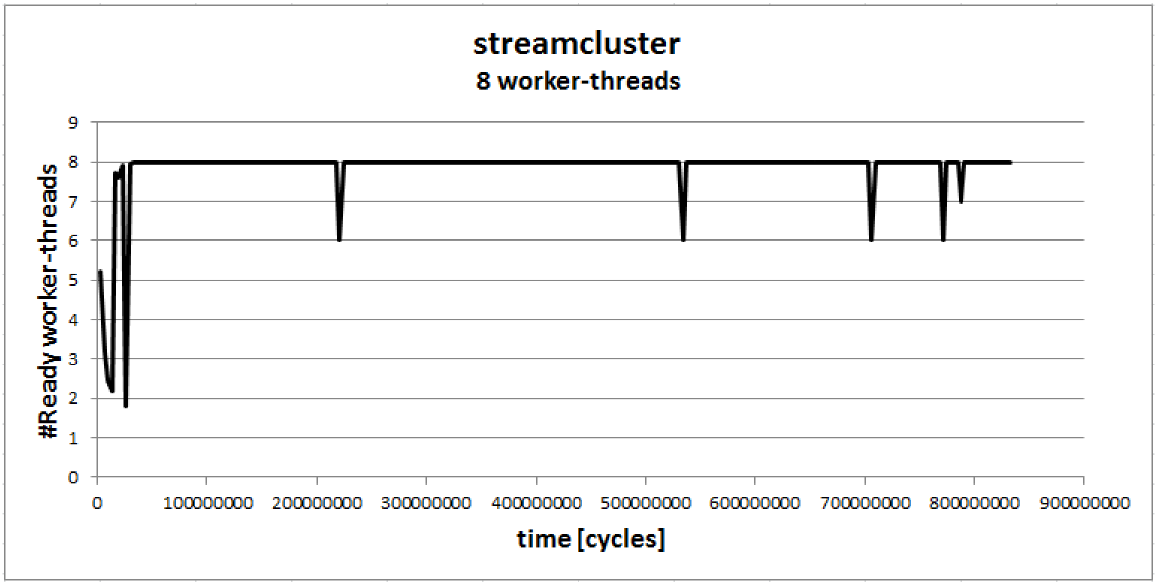
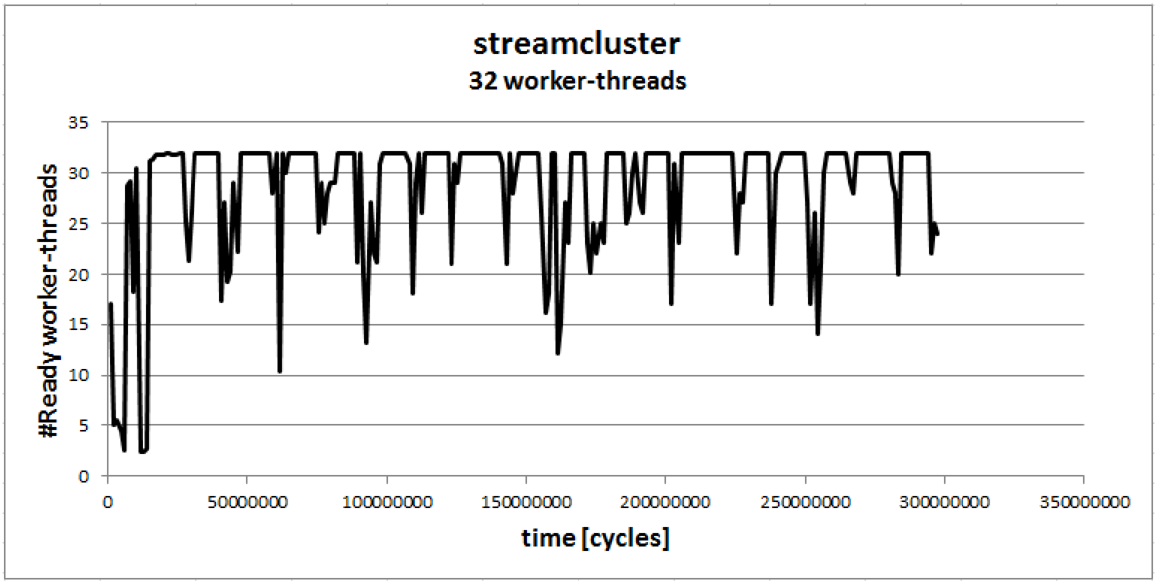
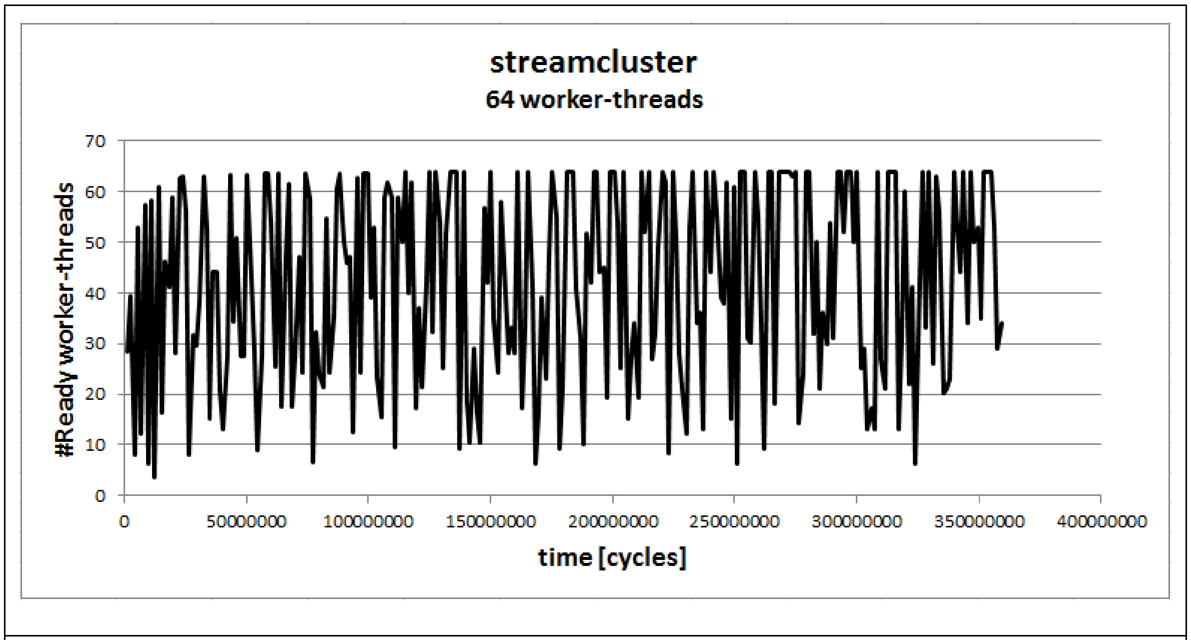


Figure 4-24: Parallelism scalability – streamcluster

The running threads over-time curves are shown in Figure 4-25. They show that as the number of worker threads increases the periods that all of them are running decreases but on average the number of running threads is in the order of the worker number of worker threads, which results good scalability.


streamcluster
8 worker-threads
#Ready worker-threads
time [cycles]
0
1
2
3
4
5
6
7
8
9
0
100000000
200000000
300000000
400000000
500000000
600000000
700000000
800000000
900000000


streamcluster
32 worker-threads
#Ready worker-threads
time [cycles]
0
5
10
15
20
25
30
35
0
50000000
100000000
150000000
200000000
250000000
300000000
350000000



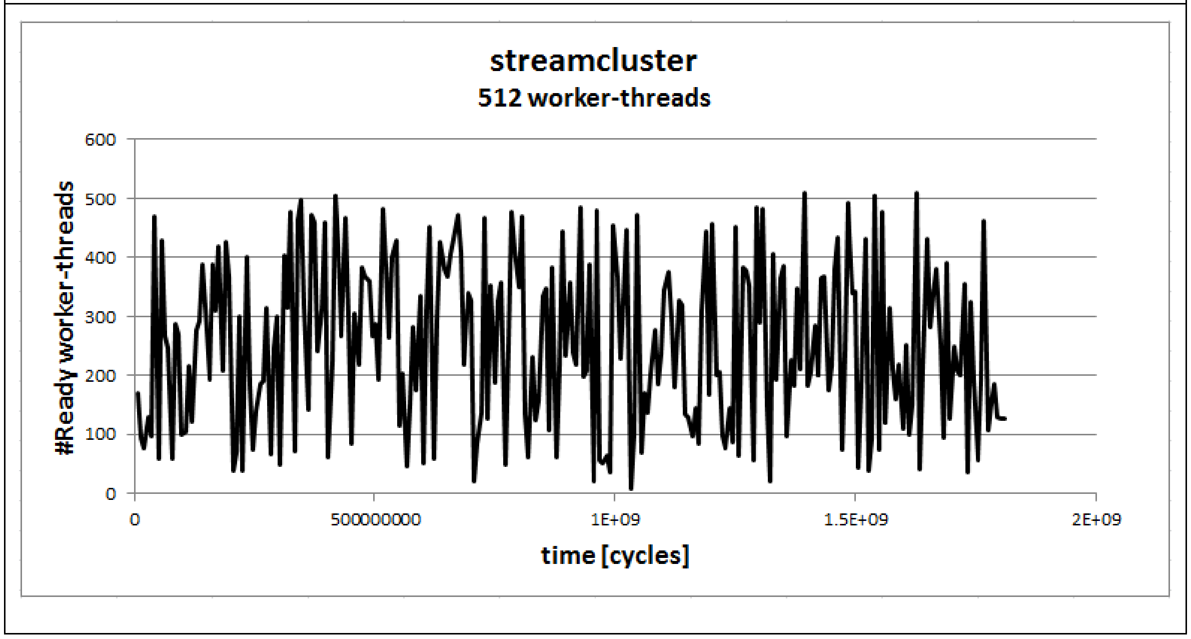


Figure 4-25: Running threads over time - streamcluster

4.3.10. swaptions

The performance of the **swaptions** benchmark is shown in Figure 4-26. It shows perfect parallelism up to ~320 threads, and then it virtually plateaus up to 1024 threads, then increases and plateaus again at ~1152 threads.

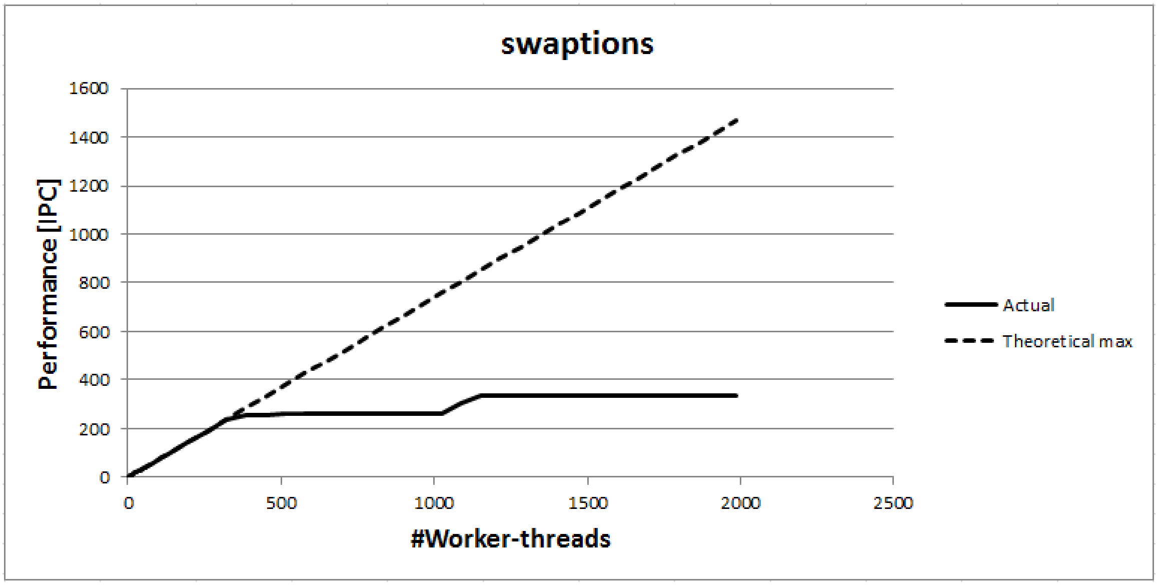
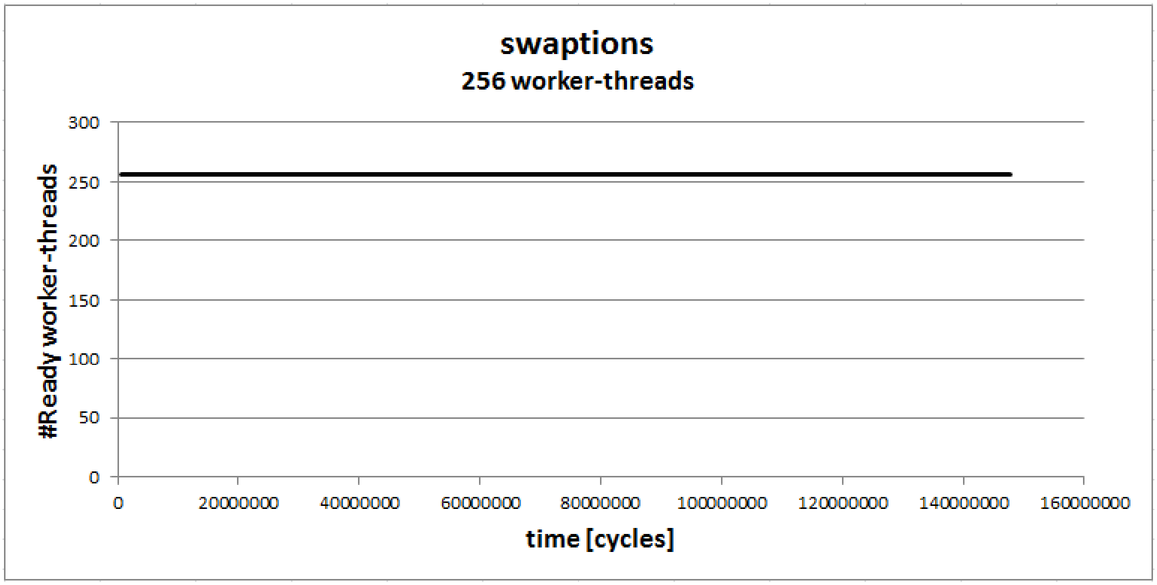
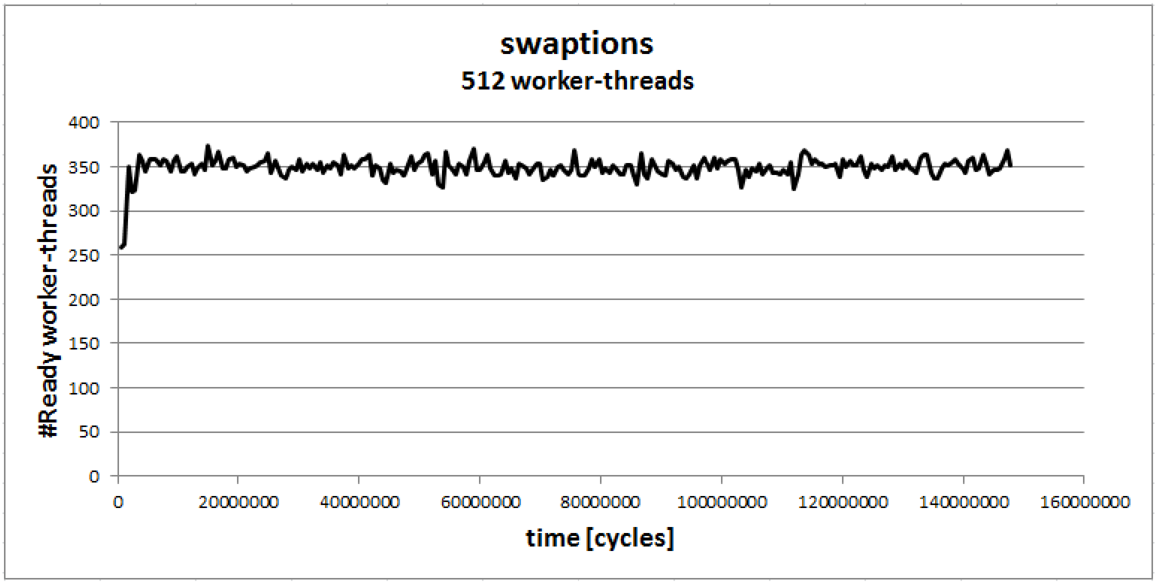


Figure 4-26: Parallelism scalability - swaptions

The **swaptions** benchmark is *embarrassingly parallel* – it distributes the data-set evenly between the worker-threads, which operate on them independently. However, unlike **blackscholes** which is also embarrassingly parallel this benchmark doesn't exhibit perfect parallelism throughout the worker-threads count spectrum. This is because this benchmark dynamically allocates memory throughout its parallel execution. Thus while there is no inter-thread dependency in the algorithm, such a dependency is introduced through the dynamic memory allocation heap manager, because the heap is shared by all threads and therefore has to have some inter-thread synchronization. The running threads over-time curves in Figure 4-27 reflect that – with 256 threads the heap synchronization is negligible. With 512 and above, the computation involves in the heap management dwarfs the computation of the actual algorithm so there is excessive contention which prevents all the threads from running simultaneously. Therefore, the measured parallelism scalability is actually the heap manager's parallelism scalability rather than the **swaptions** algorithm itself.


swaptions
256 worker-threads
#Ready worker-threads
300
250
200
150
100
50
0
0
20000000
40000000
60000000
80000000
100000000
120000000
140000000
160000000
time [cycles]


swaptions
512 worker-threads
#Ready worker-threads
400
350
300
250
200
150
100
50
0
0
20000000
40000000
60000000
80000000
100000000
120000000
140000000
160000000
time [cycles]

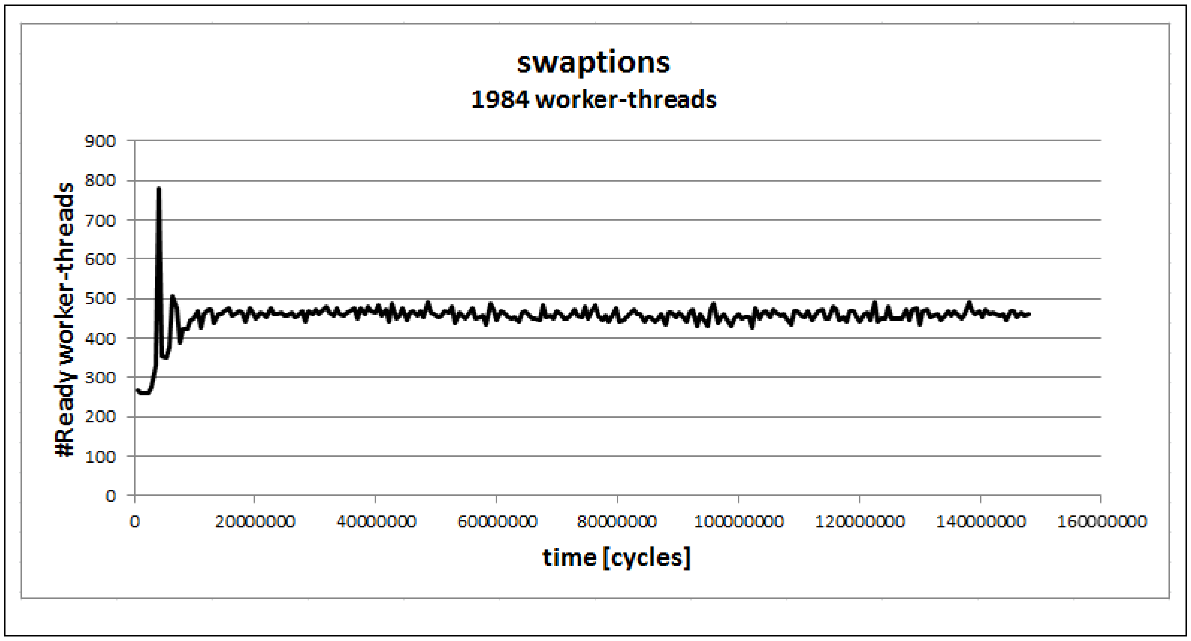


Figure 4-27: Running threads over time - swaptions

4.3.11. vips

The performance of the **vips** benchmark is shown in Figure 4-28. Figure 4-29 is similar to Figure 4-28 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. It shows excellent scalability up to 80 threads but then the performance plateaus and even slightly decreases.

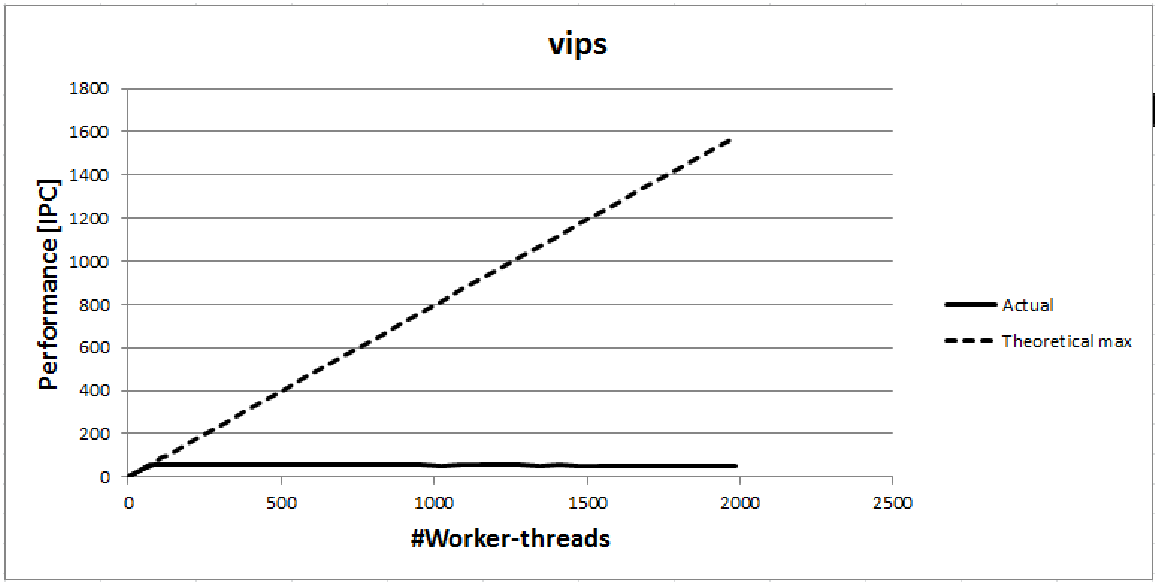


Figure 4-28: Parallelism scalability - vips

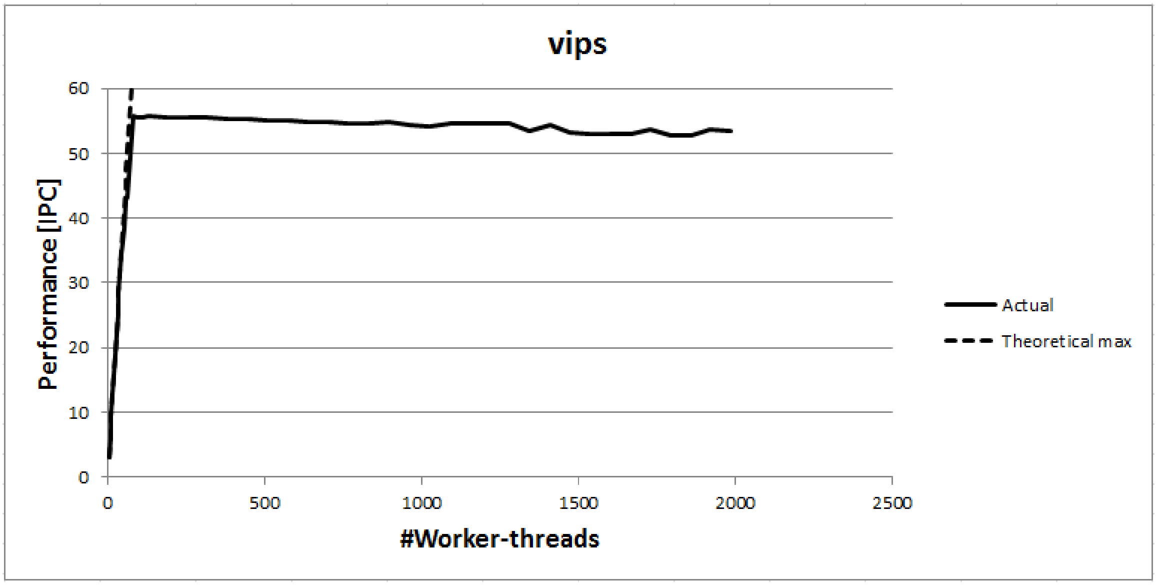
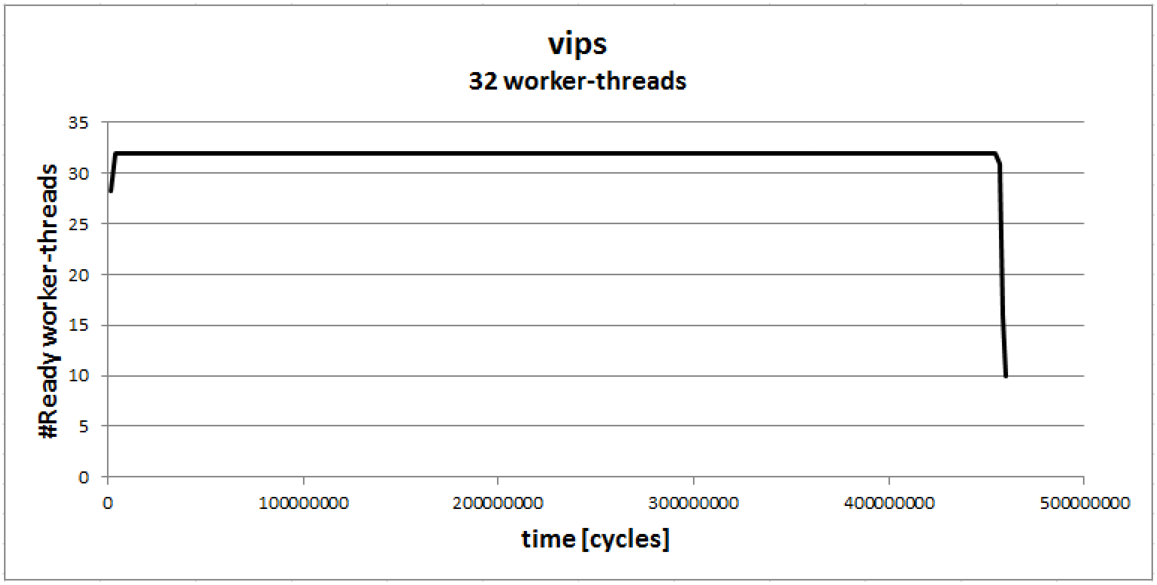
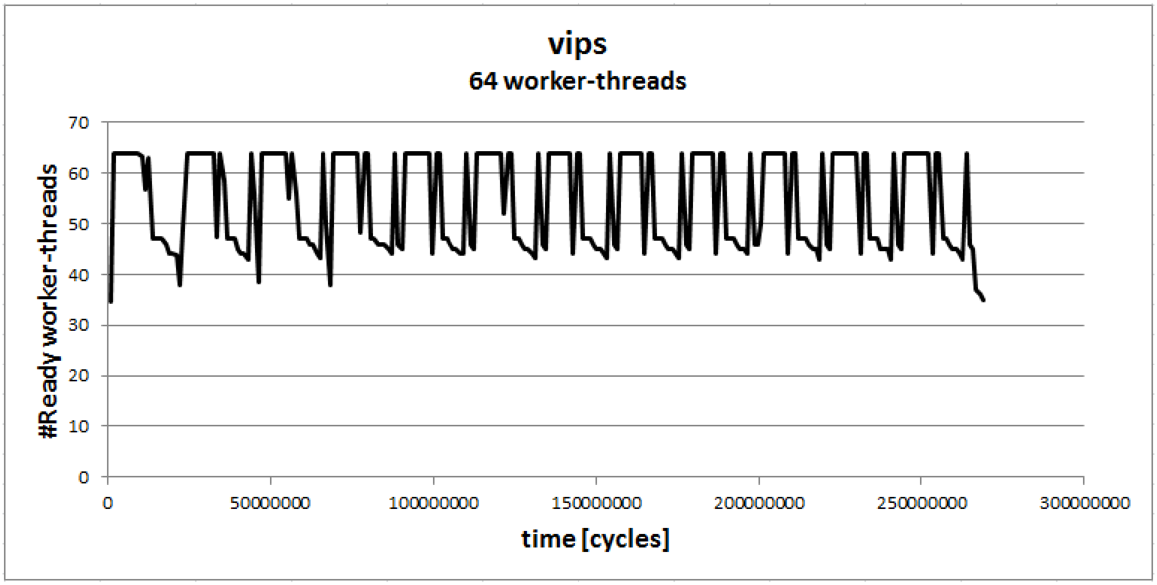
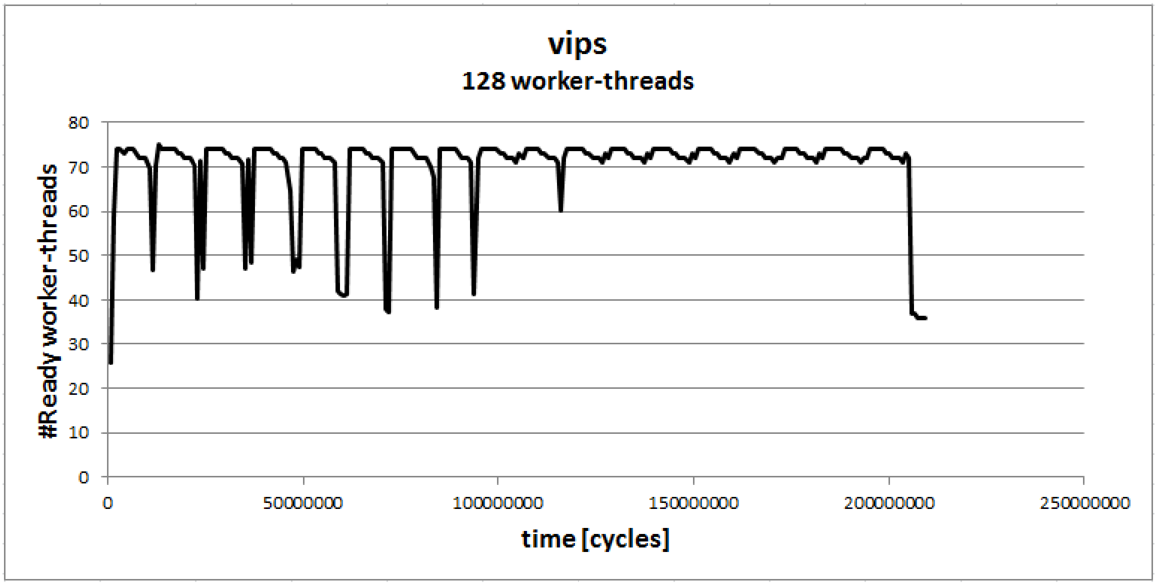


Figure 4-29: parallelism scalability – vips (scaled)

The running threads over-time curves are shown in Figure 4-30. They show that while with 64 thread or less all of the threads are running most of the time, with 128 threads and above there are never more than 75 threads that are running simultaneously.


vips
32 worker-threads
#Ready worker-threads
time [cycles]
0
5
10
15
20
25
30
35
0
100000000
200000000
300000000
400000000
500000000


vips
64 worker-threads
#Ready worker-threads
time [cycles]
0
10
20
30
40
50
60
70
0
50000000
100000000
150000000
200000000
250000000
300000000



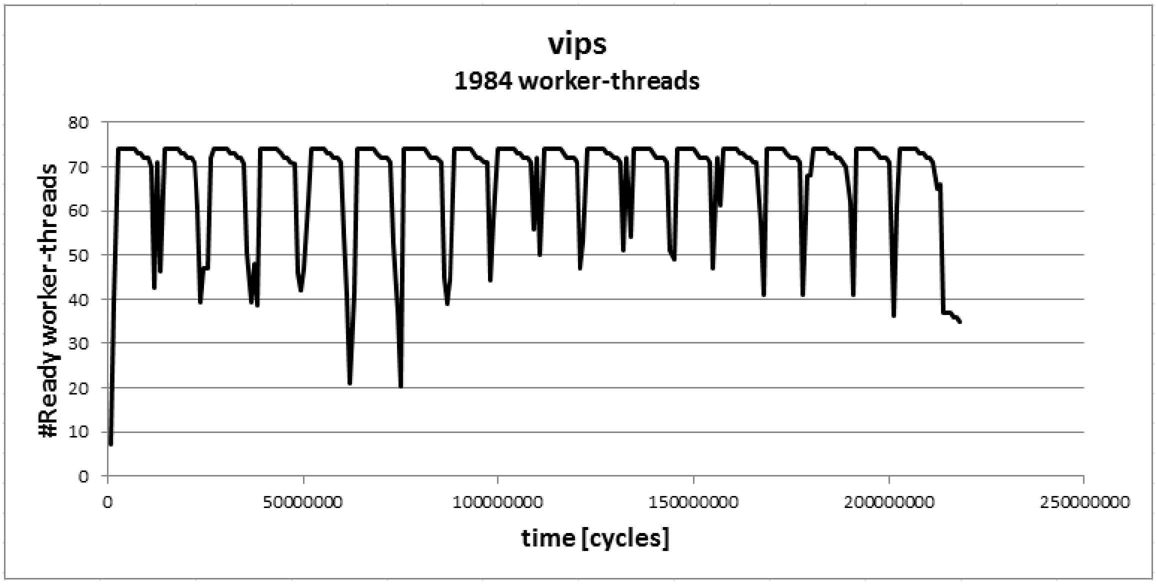


Figure 4-30: Running threads over time - vips

4.3.12. x264

The performance of the **x264** benchmark is shown in Figure 4-31. Figure 4-32 is similar to Figure 4-31 except that the vertical axis is scaled, to provide a more detailed view of the actual performance curve. The performance plateaus at 8 threads with a decrease at 36 threads.

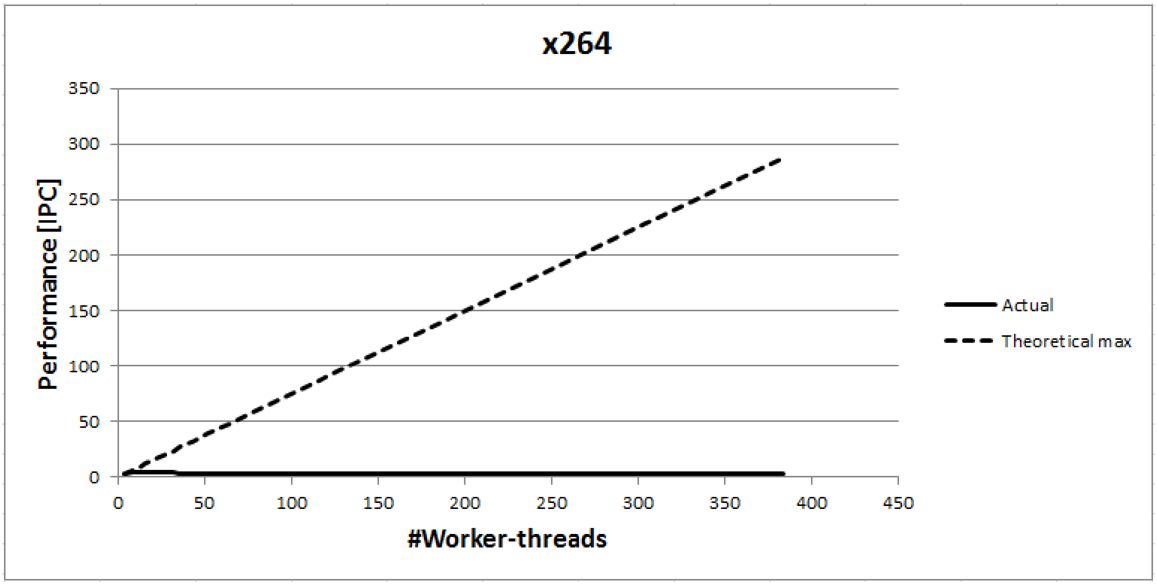


Figure 4-31: Parallelism scalability - x264

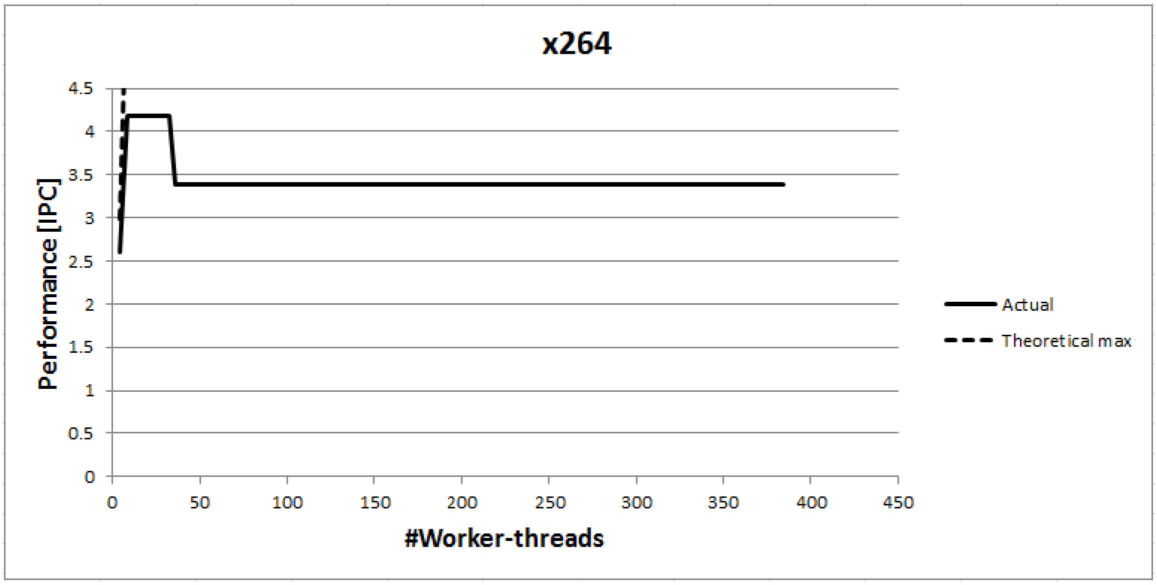
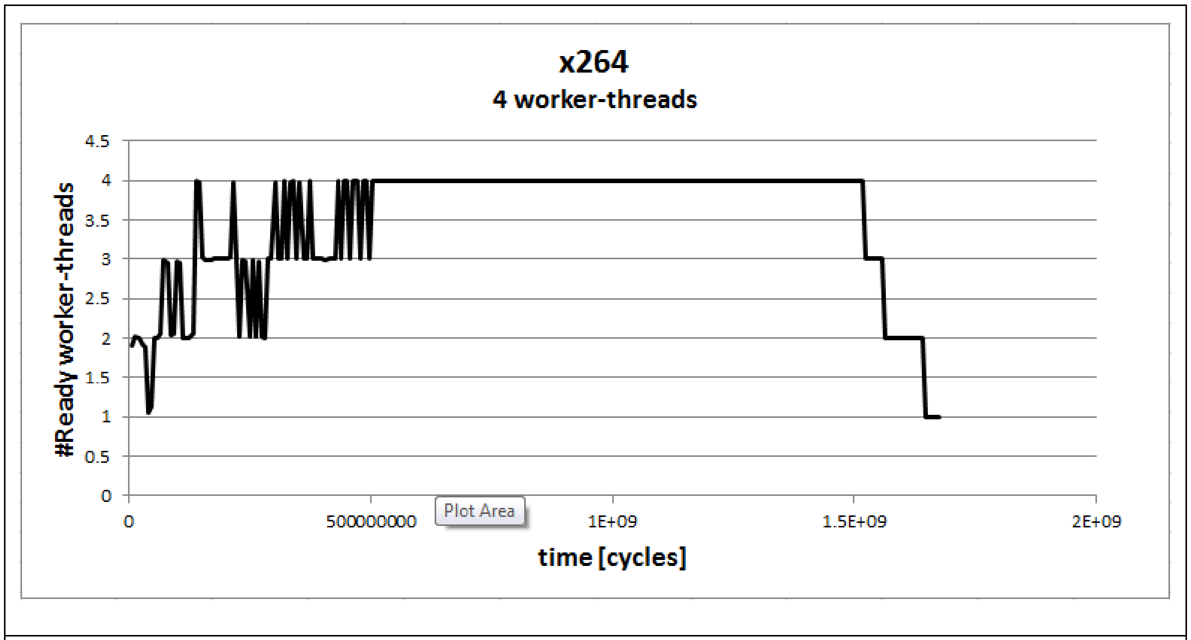
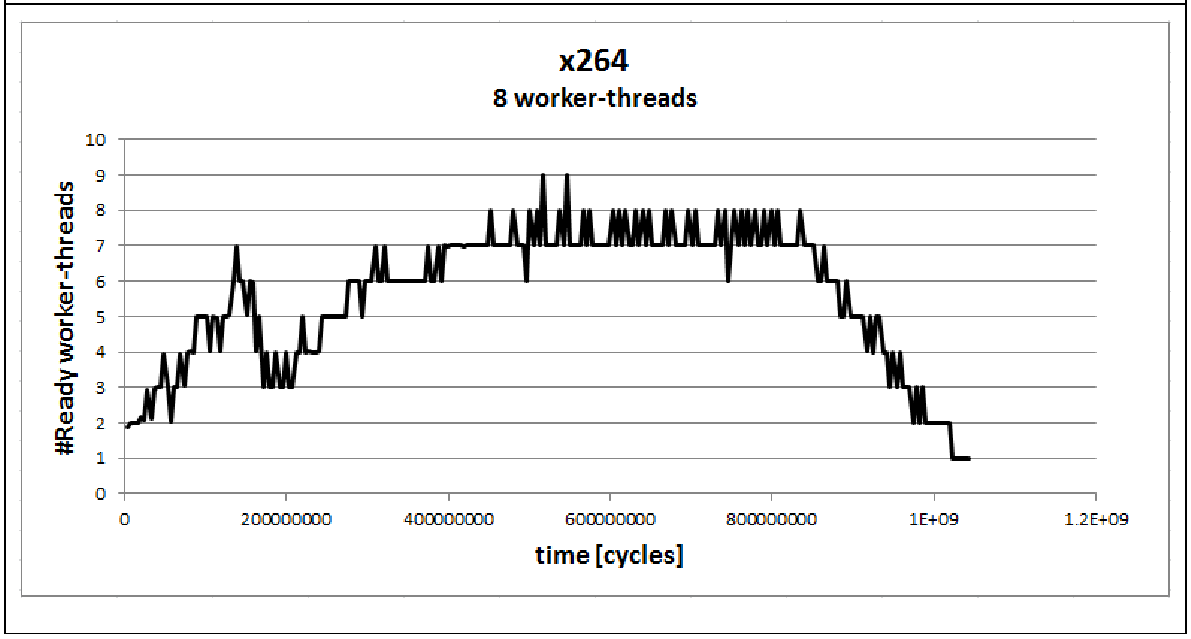


Figure 4-32: Parallelism scalability - x264 (scaled)

The running threads over-time curves are shown in Figure 4-33. They show that there are never more than 9 threads running simultaneously and the execution pattern from 64 threads and above is the same.


x264
4 worker-threads
#Ready worker-threads
time [cycles]
0
500000000
1E+09
1.5E+09
2E+09
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Plot Area


x264
8 worker-threads
#Ready worker-threads
time [cycles]
0
200000000
400000000
600000000
800000000
1E+09
1.2E+09
0
1
2
3
4
5
6
7
8
9
10

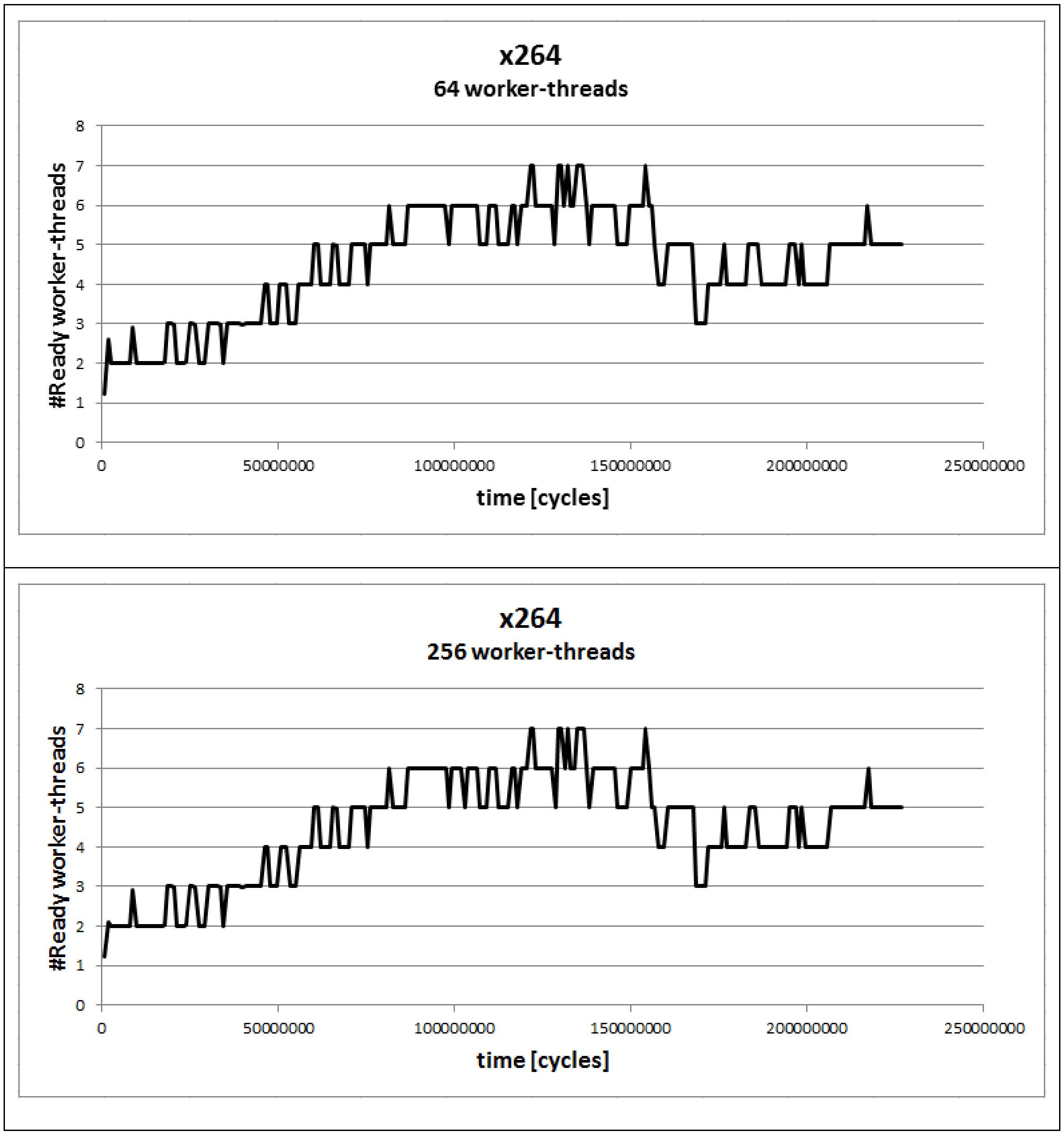


Figure 4-33: Running threads over time - x264

4.3.13. freqmine

The **freqmine** benchmark doesn't support parallelism through pthreads [17], only OpenMP [18]. In particular, there is no notion of ROI when executed in the OpenMP mode. Therefore, we did not study this benchmark.

4.4. Conclusions

A principal limiting factor of workload's ability to utilize parallel architectures is its ability to partition the computation into enough independent tasks throughout its execution. If not, the parallel processing elements that are available in the architecture cannot all be fully utilized, implying mismatch between the workload and the architecture.

We study this ability of given workloads to partition the computation into enough tasks through measuring their performance on a parallel-perfect architecture, i.e. one with no parallelism limitation: as much processing elements as there are threads and no shared resources that parallel tasks may contend over. Thus, any deviation from full utilization of the processing elements is necessarily due to the workload, not the architecture. Such deviation necessarily stems from inter-thread synchronization, where a thread waits for another thread to complete an operation, hence cannot perform any computation, and thus cannot utilize the processing element that is presumably available in the underlying architecture.

From the performance measurements we make the following conclusions:

1. Most Parsec workloads have parallelism degrees that vary significantly during their execution, i.e. their parallelism behavior is not stationary in time.
2. A workload that instantiates a lot of worker threads cannot necessarily make all of them indeed work in parallel – the maximum number of simultaneously active threads may never reach the number of worker threads.
3. Even when a workload is able to make all the worker threads active simultaneously, the performance gain may be negligible relative to smaller number of worker threads because the duration of this phase of peak utilization may be small relative to the total execution time. For one benchmark (canneal) beyond a certain number of worker threads the performance even degrades (on a parallel-perfect architecture!).
4. The performance of most of the benchmarks in the Parsec suite peak at largely varying numbers of worker threads, implying large differences in parallel scaling capabilities. Table 4-2 shows the peak parallelism degree for each benchmark, i.e., the parallelism degree beyond which the marginal gain in

further increase in the parallelism degree of the underlying architecture (and corresponding increase in the number of worker threads) is negligible. It is important to note that the architecture's parallelism degree referred to here is the number of cores, not necessarily the number of threads. For example, with a multi-threaded architecture, i.e., with more thread-contexts than cores and a hardware scheduler that switches threads when a thread is stalled waiting for long RAM access, the performance can be improved by having as many threads as there are thread-contexts, which is more than the number of cores, because the additional threads can utilize core's cycles that would be idle if there were as many cores as threads.

Benchmark	Minimum #threads and #cores for peak performance
blackscholes	≥1984
bodytrack	32
canneal	128
dedup	96[6]
facesim	32
ferret	4
Fluidanimate	≥1024
raytrace	80
streamcluster	≥640
swaptions	320[7]
vips	80
x264	8

Table 4-2: Parsec per-benchmark inherent parallelism limitations

[6] This benchmark plateaus when invoked with 64 threads but it actually spawns 3 times this number of threads and gets up to ~1.5 times running threads than the number of threads in the invocation, thus it can utilize up to $1.5 \cdot 64 = 96$ cores

[7] The limiting factor is the heap. Thus, a different heap implementation may result substantially different limitations

5. The inter-thread dependency is captured by the graph of running threads over time. This is useful for directing parallelism-oriented performance optimization.

   The graph of running threads over time is not easily obtained on actual hardware – the actual collection of the data could distort the results, in particular if the data, which may be huge, needs to be written to external store (e.g., disk or network store). Thus, simulation is highly useful for detailed study of this aspect of parallel workloads.

Chapter 5. Cache analytical modeling study

5.1. Introduction

In this chapter we use our simulator to extract the cache performance (miss-rate) of the benchmarks in the Parsec benchmark suite with various degrees of parallelism (up to the maximum supported by the simulator and the specific benchmark) and compare it with the analytical cache performance model proposed in [19]:

$$\textbf{(5-1)} \qquad \mathbf{P_{miss}(S_\$)} = \left(\frac{S_\$}{\beta} + 1\right)^{-(\alpha-1)}$$

Where $S_\$$ is the cache size and α and β are parameters that depend on the workload. This model is based on the well-known empirical power law from the 70's (also known as the 30% rule or the √2 rule) [20]. In Equation (5-1), workload locality increases when increasing α or decreasing β.

[4] proposes a simple adaptation of (5-1) to parallel workloads: it is assumed that the threads do not share data, in which case the cache store space is effectively divided between the threads. In other words, the effective per-thread cache size (EPTCS henceforth, for brevity) is the total size of the shared cache $S_\$$ divided by the number of threads n. Incorporating this into (5-1), we get formula (5-2).

$$\textbf{(5-2)} \qquad \mathbf{P_{miss}(S_\$, n)} = \left(\frac{S_\$/n}{\beta} + 1\right)^{-(\alpha-1)}$$

In (5-2) the parameter β may also account for the degree of sharing among the threads: in case much of the cache is shared, each thread can utilize a larger portion of the cache, which is represented by a smaller value of β.

While (5-2) is a two-dimensional function, we notice that it maintains equation (5-3) and thus can be expressed as one-dimensional function. Therefore, for our analysis and graphical representation we use this one-dimensional form of the miss-rate function.

(5-3) $$\mathbf{P_{miss}}(\mathbf{S_\$}, \mathbf{n}) = \mathbf{P'_{miss}}\left(\frac{\mathbf{S_\$}}{\mathbf{n}}\right)$$

5.2. Methodology

We extract the actual miss-rate functions of the various benchmarks in the Parsec benchmark suite through simulation with different cache sizes and different parallelism degree. The simulated parallelism degrees are the same as the ones used in the parallelism scalability study in Chapter 4. Therefore, the simulations used for the study of the cache performance use the same number of worker-threads, data sets, code modifications and ROI detection mode as used for the parallelism scalability study simulations.

Using the number of worker-threads for *n* in (5-2) is not appropriate because not all worker-threads are necessarily running all the time and therefore not all necessarily compete for the cache storage space. In particular, if a workload instantiate many worker threads but only a small number of them is active at any given time, then only those that are active compete for the cache[8]. To reflect this, *n* is set to the *average number of running threads*, as extracted from the simulation.

Table 5-1 summarizes the parameters of the simulation model that is used to study the cache performance. As shown, the benchmarks are simulated with different cache sizes. The cache-model that we use is 64-way set-associative with 64 bytes per cache-line, and classical per-set Least-Recently-Used (LRU) replacement policy.

Note that the simulation model for this shared cache study does not include private cache.

[8] It should be noted that in theory it is possible that the competition on the cache storage has the effect of all the worker threads *n* competing even though only a small number of threads is active at any given time – when each thread works for a very short time *t* and blocks for an order of $n \cdot t$. However, we consider this to be unlikely, especially with high degree of parallelism because it implies excessive inter-thread synchronization.

Parameter	Description
N_{PE}	>= #worker-threads
$S_\$$	512KB, 1MB, 2MB, 4MB, 8MB, 16MB
N_{max}	N_{PE}
CPI_{exe}	1 *[cycles]*
$t_\$$	1 *[cycles]*
t_m	200 *[cycles]*

Table 5-1 Model parameters for the cache modeling study

For every benchmark we show the miss-rate curves vs. EPTCS for each cache size separately in a single graph. The miss-rate may or may not be sensitive to the absolute cache size. When the miss-rate is not sensitive to the cache size, i.e., affected only by EPTCS, the graphs of the different cache sizes overlap.

In a separate graph we show all the data-points from all cache sizes and a curve of formula (5-2) that is fitted to these data-points. For the fitted curve we need to find α and β that would result a curve that is closest to the data-points. We do the fitting using the Levenberg–Marquardt curve-fitting algorithm [21], specifically the *lmfit* open-source implementation of that algorithm [22].

Given the actual and fitted miss-rates, we derive the respective analytical performance as a function of the average number of running threads using formula (2-5). The performance derived from the fitted miss-rate is the performance that is predicted by the analytical performance model (section 2.2). We show the actual and fitted performance graphs to compare the actual and predicted performance.

The magnitude of the difference between the actual and predicted performance obviously corresponds to the magnitude of the difference between the actual and fitted miss-rate. To capture the effect of differences in miss-rates on differences in performance we define the *performance sensitivity to miss-rate* as the marginal change in performance on changes in miss-rate. This is captured by the derivative of the performance formula (2-5) with respect to the miss-rate, shown in formula (5-4). As can be seen, the sensitivity depends on the model parameters' values. In particular, the sensitivity is proportional to the number of threads *n*.

$$\text{(5-4)} \quad \mathbf{Performance}'(\mathbf{S}_\$, \mathbf{n}) = -\frac{\mathbf{n} \cdot \mathbf{r_m} \cdot (\mathbf{t_m} - \mathbf{t_\$})}{\left(\mathbf{CPI_{exe}} + \mathbf{r_m} \cdot \left(\left(1 - P_{miss}(S_\$, n)\right) \cdot t_\$ + P_{miss}(S_\$, n) \cdot t_m\right)\right)^2}$$

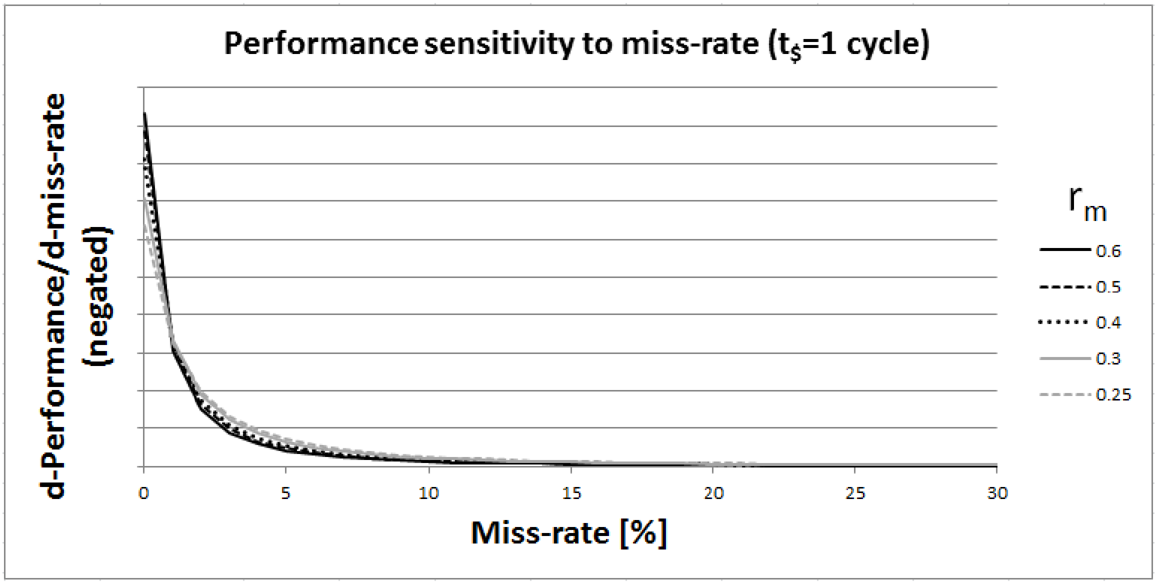


Figure 5-1: Performance sensitivity to miss-rate[9] for different values of r_m

Figure 5-1 depicts the performance sensitivity to miss-rate for different values of r_m when $t_\$$ is fixed (we use 1 cycle, because this is what was used in our simulations). The values used for r_m are between the minimum and maximum values that were measured in our simulations – 0.25 and 0.6, respectively. This graph shows that there is little difference in sensitivity across these values of r_m.

Figure 5-2 is a graphical depiction of the performance sensitivity to miss-rate for different values of $t_\$$ when r_m is fixed (we use the value 0.425, which is in the middle between the minimum and maximum values that were measured in our simulations). This

[9] The actual values on the vertical axis are not shown because they are proportional to *n* and the units are not particularly meaningful (*instructions-per-cycle-per-miss-rate*). Hence this graph reflects the relative differences of the sensitivity between different values of r_m.

shows monotonously increasing sensitivity as the miss-rate decreases, with the increase in sensitivity being steeper with smaller values of $t_\$$. This means that a difference between the fitted and actual miss-rates results increasingly larger difference between the actual and predicted performance as the actual miss-rate becomes smaller. However, since the sensitivity is proportional to the number of threads, the latter is a principal factor of the sensitivity.

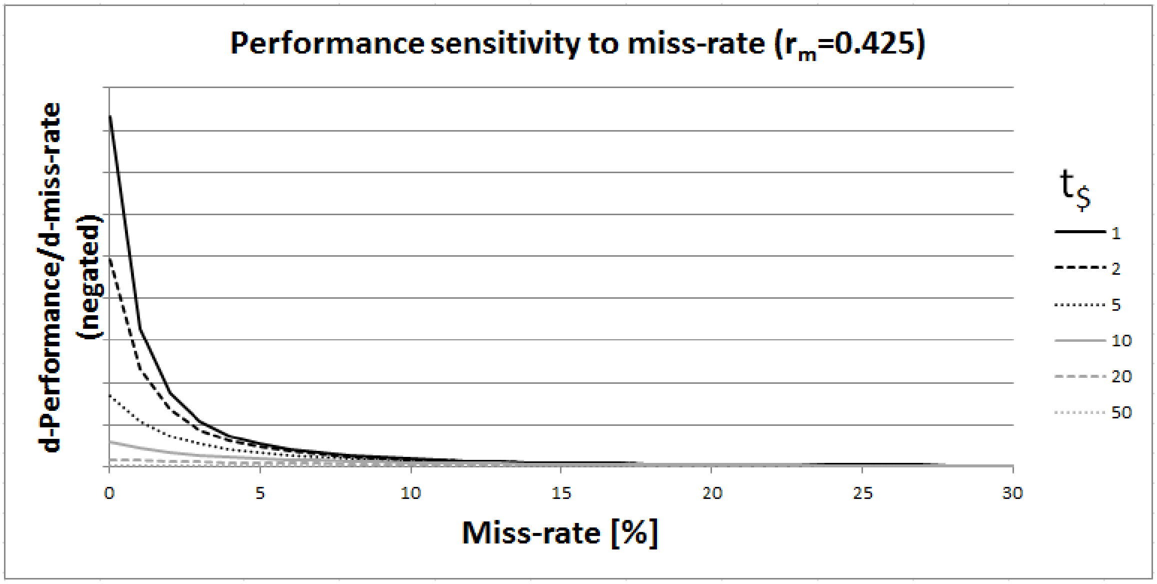


Figure 5-2: Performance sensitivity to miss-rate for different values of $t_\$$

5.3. Simulation results

5.3.1. Blackscholes

Figure 5-3 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

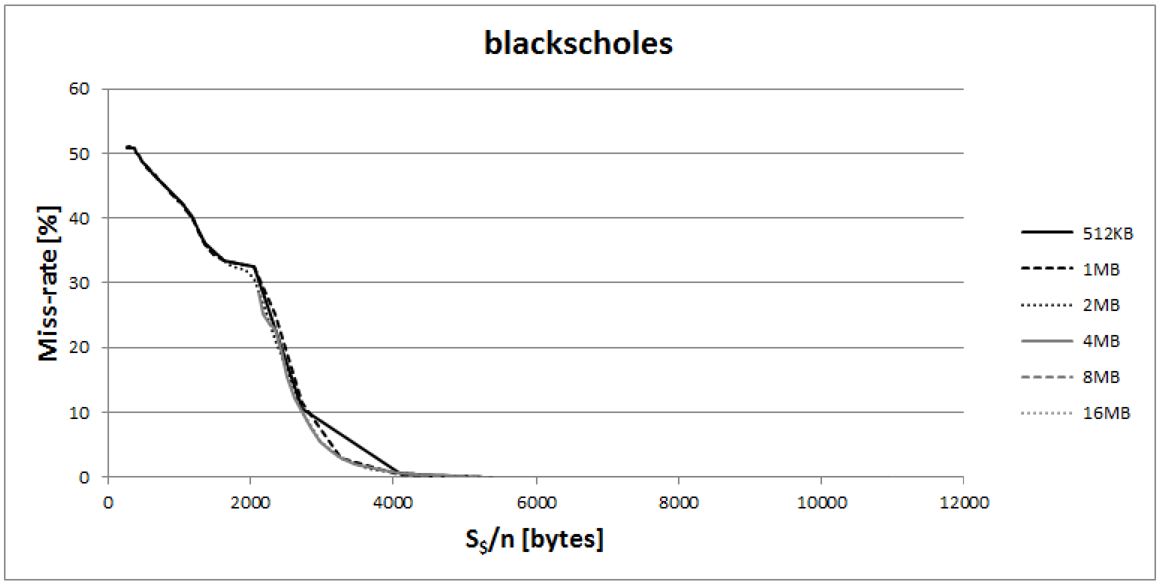


Figure 5-3: Miss-rate from simulation – blackscholes

Figure 5-4 shows all the actual miss-rate data-points, for all cache sizes and all thread counts as measured by the simulation. It also shows the graph of the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. It shows that while the general shape is somewhat similar, the differences are quite large – around 20% at the lower region of the horizontal axis and more than 10% around 2KB on the horizontal axis.

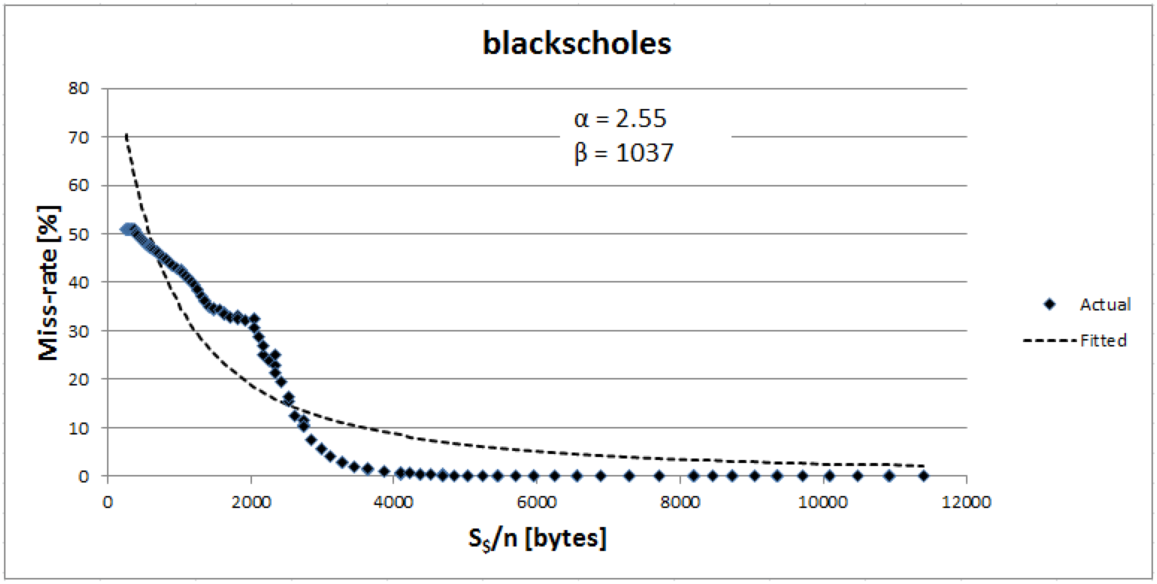


Figure 5-4: Miss-rate model fitting – blackscholes

Figure 5-5 shows the comparison of the actual performance obtained from simulation and the performance that is predicted by the analytical model using the workload parameters (Table 2-2) obtained from the same simulation. For smaller cache sizes the actual performance has a valley but the predicted performance is monotonously increasing. As the cache size increases, the peak performance is achieved at higher thread counts. The relative difference between the actual and the predicted performance is quite large – up to 4x. The peak difference is around EPTCS of 4KB, where the actual miss-rate becomes very small, implying high sensitivity to miss-rate differences, as described in Figure 5-2.

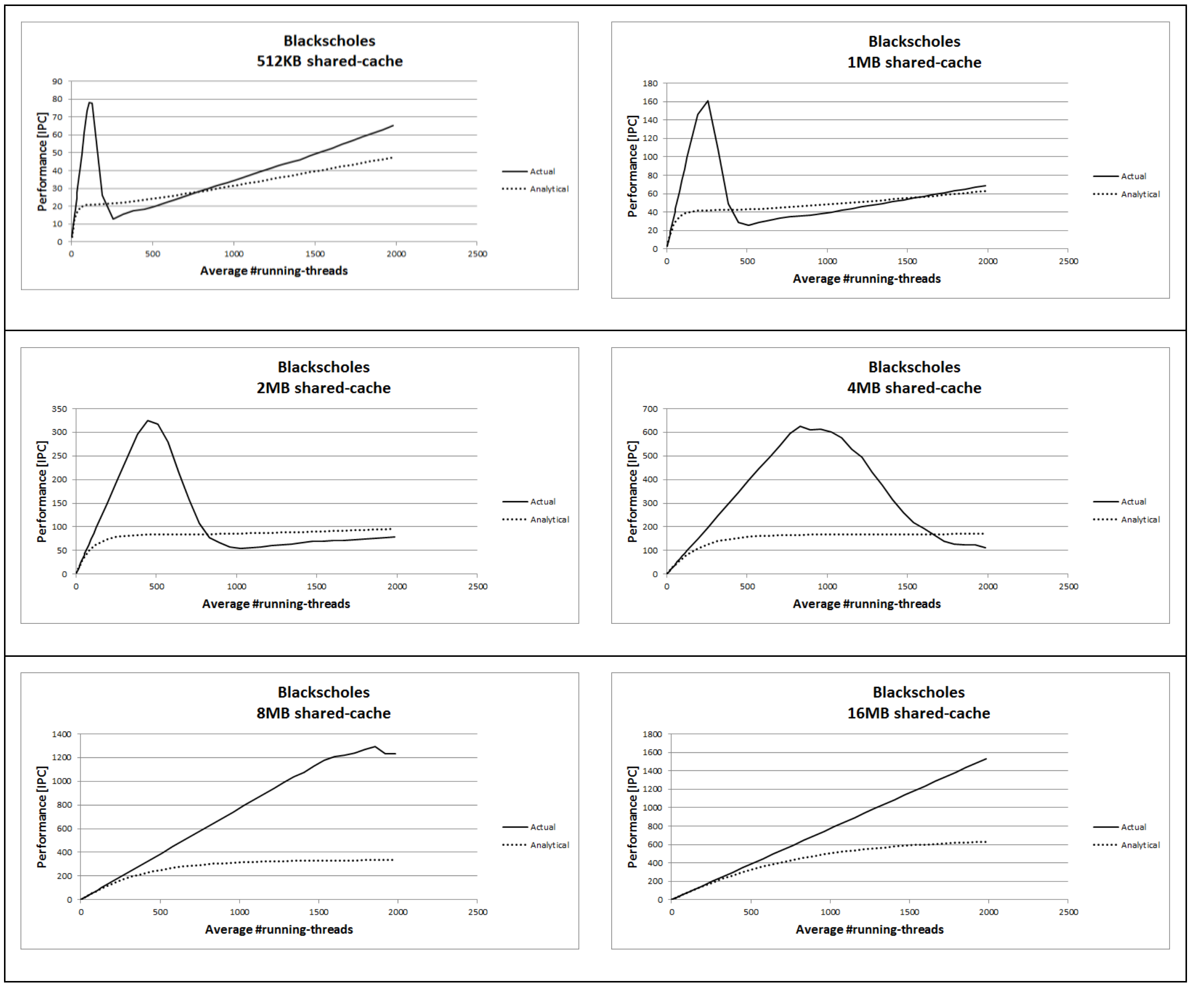


Figure 5-5: Actual vs. analytical performance with different cache sizes - blackscholes

5.3.2. Bodytrack

Figure 5-6 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are mostly not overlapping, indicating that for this benchmark the miss-rate is sensitive to the total cache size, not only to EPTCS.

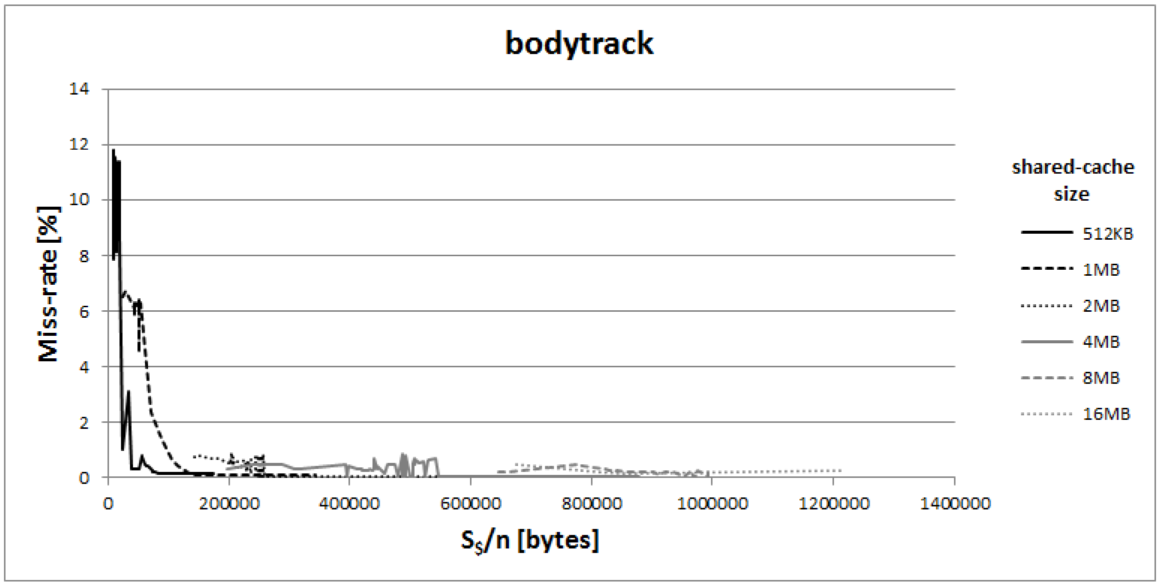


Figure 5-6: Miss-rate from simulation – bodytrack

Figure 5-7 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. It shows that while the general shape is somewhat similar, the differences are significant – up to 5% around EPTCS of 25KB on the horizontal axis. Moreover, due to the sensitivity to total cache size, the data-points of the actual miss-rate are not monotonously decreasing – similar EPTCS for different total cache size result significantly difference miss-rate.

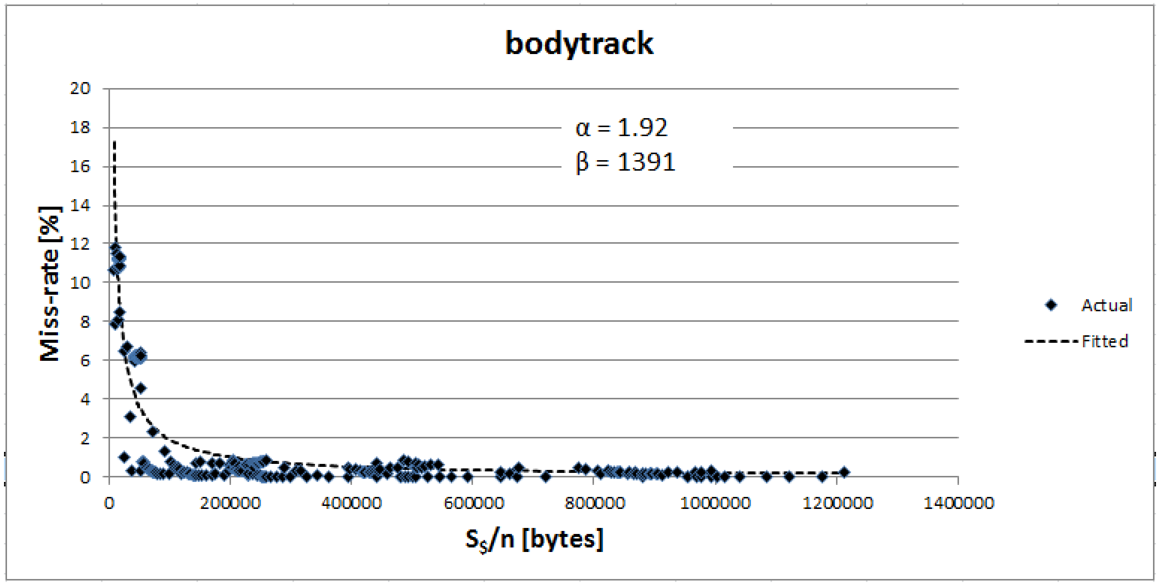


Figure 5-7: Miss-rate model fitting – bodytrack

Figure 5-8 shows the comparison of the actual performance and the performance that is predicted by the analytical model. For larger cache sizes the graphs are quite close, indicating that the analytical model provides good performance prediction. These performance values correspond to EPTCS of 140KB and more, where the difference between the actual and fitted miss-rates is relatively small. With the smaller caches, the differences are larger, with the largest being at ~22 threads with 512KB cache, which corresponds to EPTCS of ~23KB, which is the region where the difference between the actual and fitted miss-rates is the largest, with some data-point being very small, i.e., the area of large sensitivity of the performance to the miss rate.

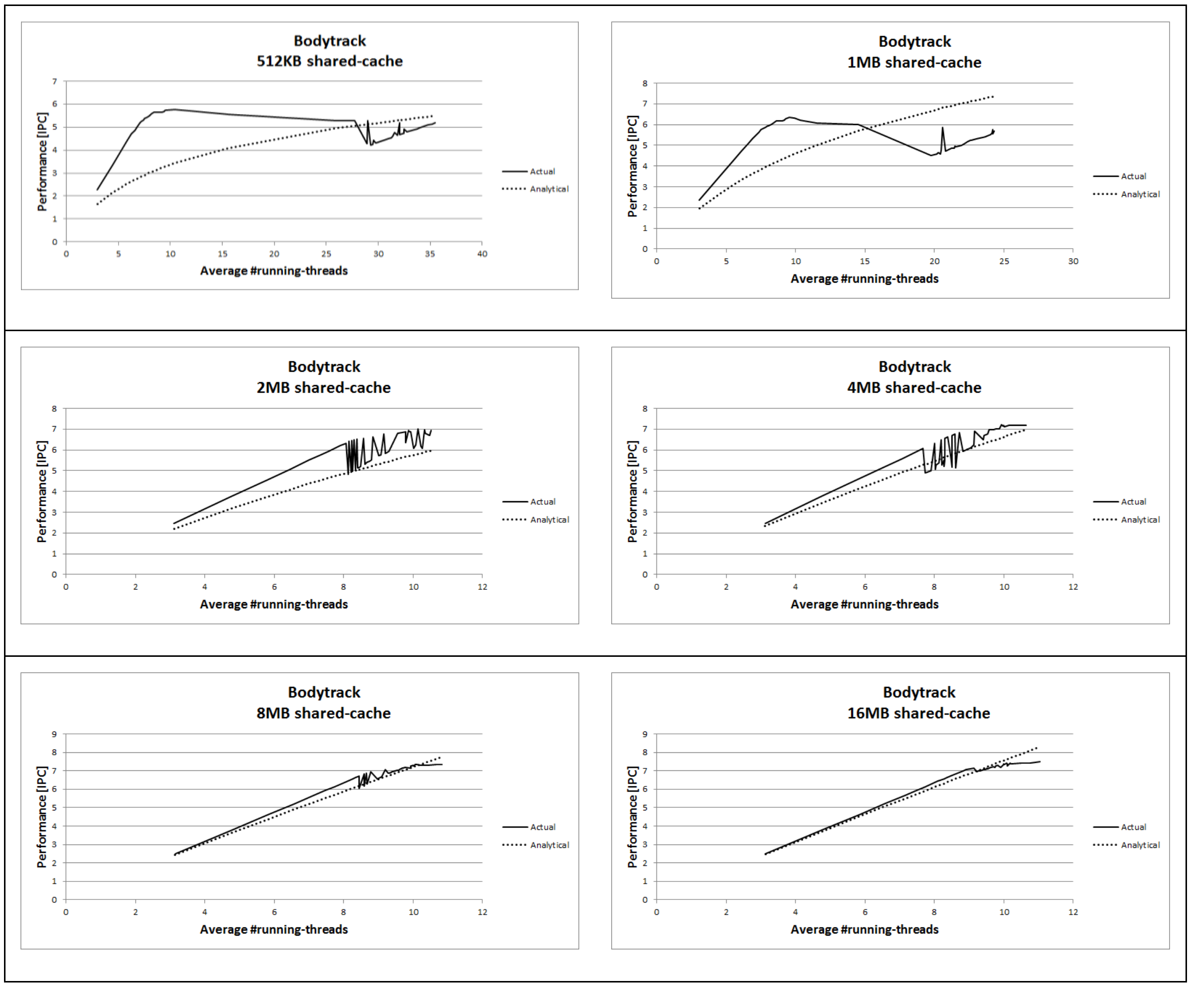


Figure 5-8: Actual vs. analytical performance with different cache sizes - bodytrack

5.3.3. Canneal

Figure 5-9 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are mostly not overlapping, indicating that for this benchmark the miss-rate is sensitive to the total cache size, not only to EPTCS. Moreover, the miss-rate alternates between higher and lower values as EPTCS increases, with the variation being bigger with smaller caches. This is the result of **canneal** having inherent negative marginal performance increase with the increase in the number of worker threads as described in section 4.3.3. Specifically, Figure 5-11 shows the average number of running threads *n* as a function of the number of worker-threads. Every graph has a region of increase and a region of decrease of *n* as the number of worker-threads increases. This means that values of *n* that are close[10] correspond to distant number of worker threads. Since the EPTCS is a function of *n*, this implies that close values of EPTCS correspond to distant number of worker-threads. Figure 4-9 shows that the **canneal** benchmark has a serial phase and a parallel phase where in the parallel phase all worker-threads are running. Therefore, all running threads compete for the cache and therefore the cache performance depends on the number of worker-threads rather than the average number of running threads.

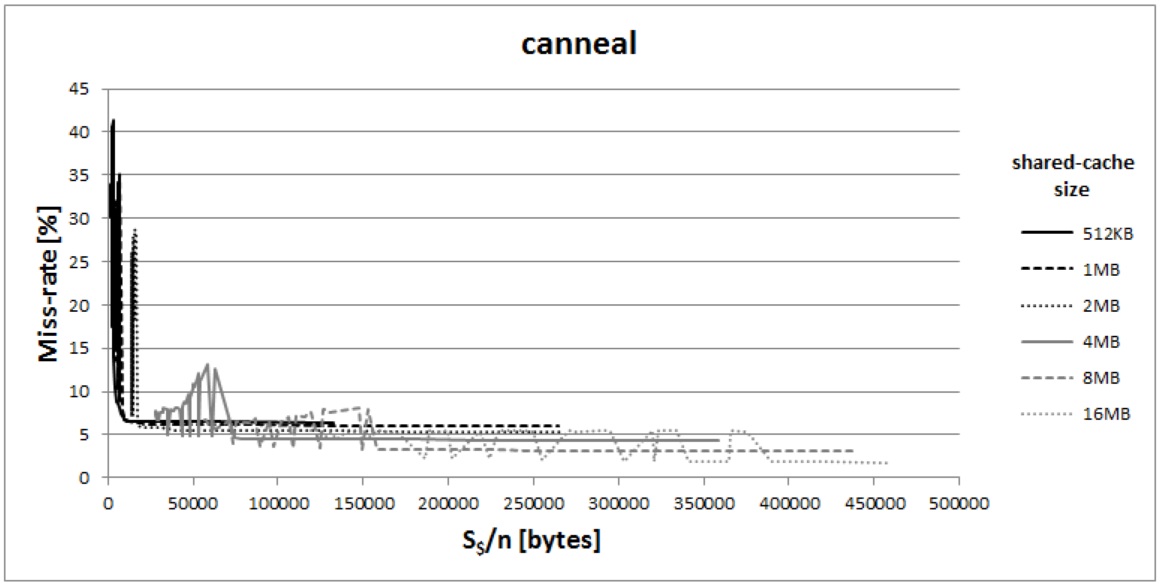


Figure 5-9: Miss-rate from simulation – canneal

[10] "Close" rather than "identical" values because n is discrete.

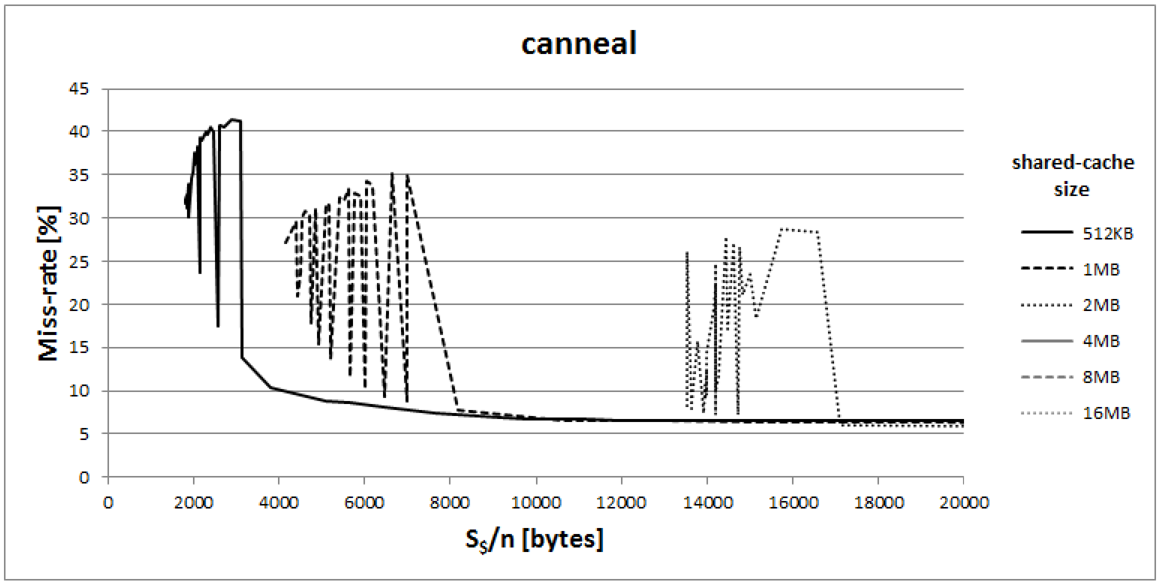


Figure 5-10: Miss-rate from simulation – canneal (zoom)

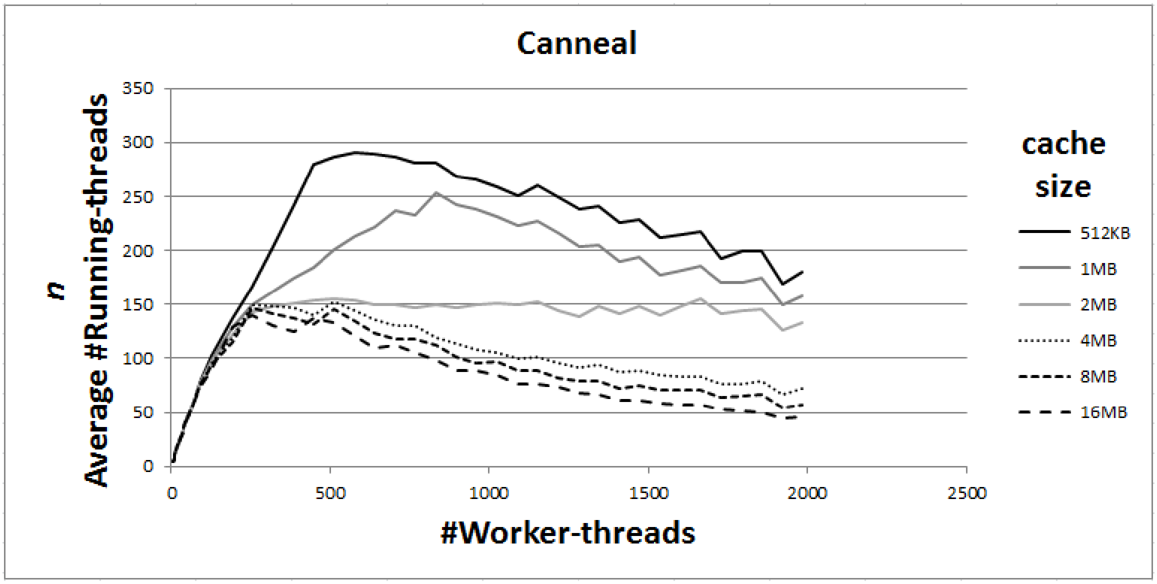


Figure 5-11: Average #running threads vs. #Worker-threads – canneal

Figure 5-12 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. It shows that while the general shape is somewhat similar, the differences are quite large. This is inevitable because the simulation data-point themselves vary a lot around the same EPTCS, as was explained.

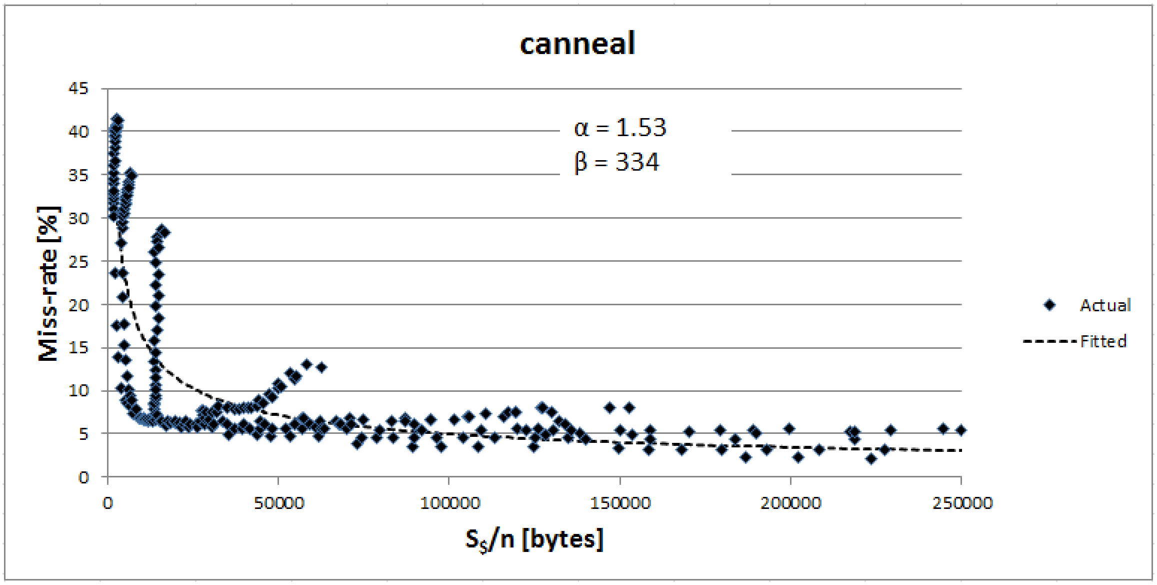


Figure 5-12: Miss-rate model fitting – canneal

Figure 5-13 shows the comparison of the actual performance and the performance that is predicted by the analytical model. The actual performance varies significantly with the average number of running threads because of the large variation in the miss-rate for close values of the average number of running threads *n*.

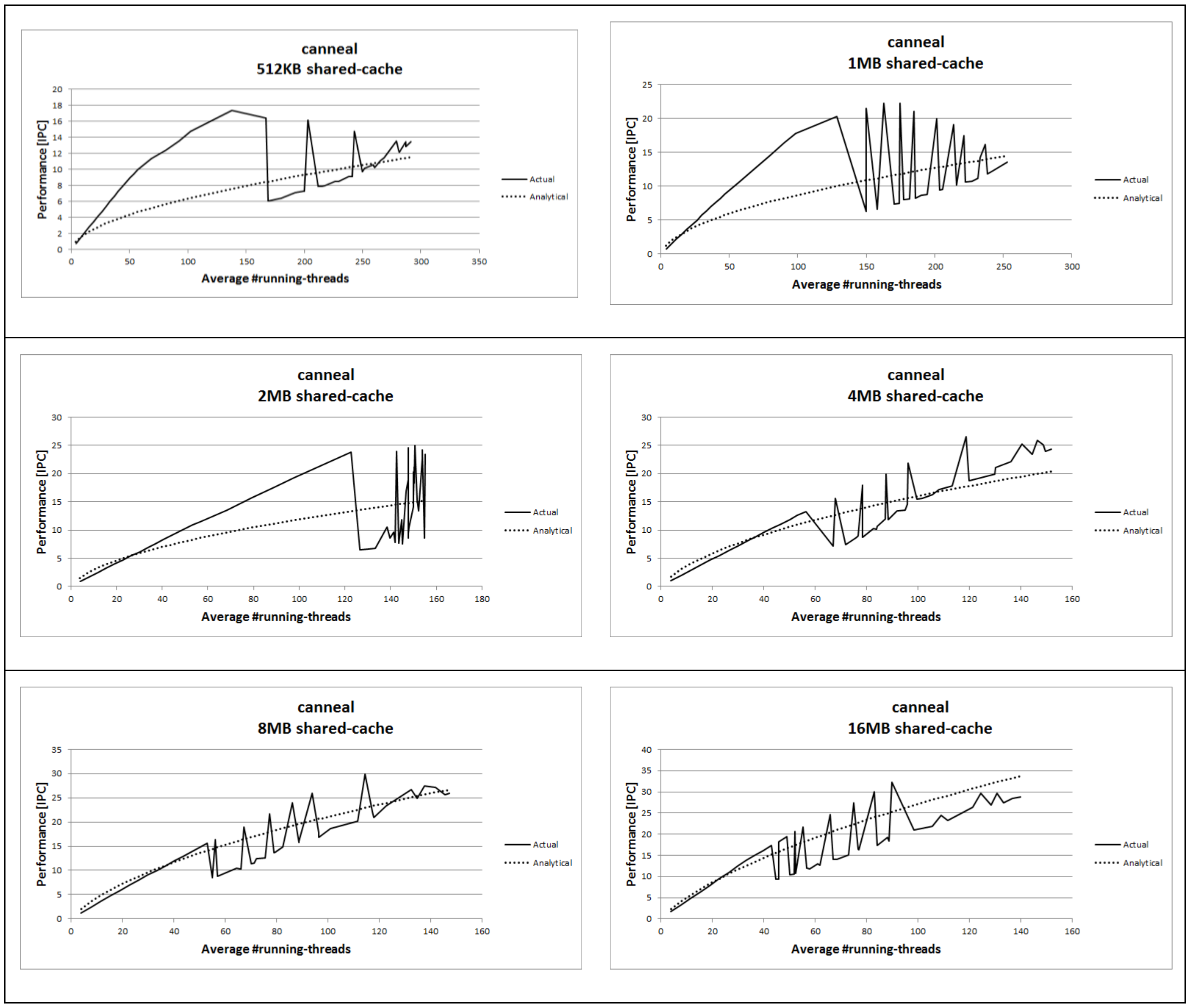


Figure 5-13: Actual vs. analytical performance with different cache sizes - canneal

5.3.4. dedup

Figure 5-14 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

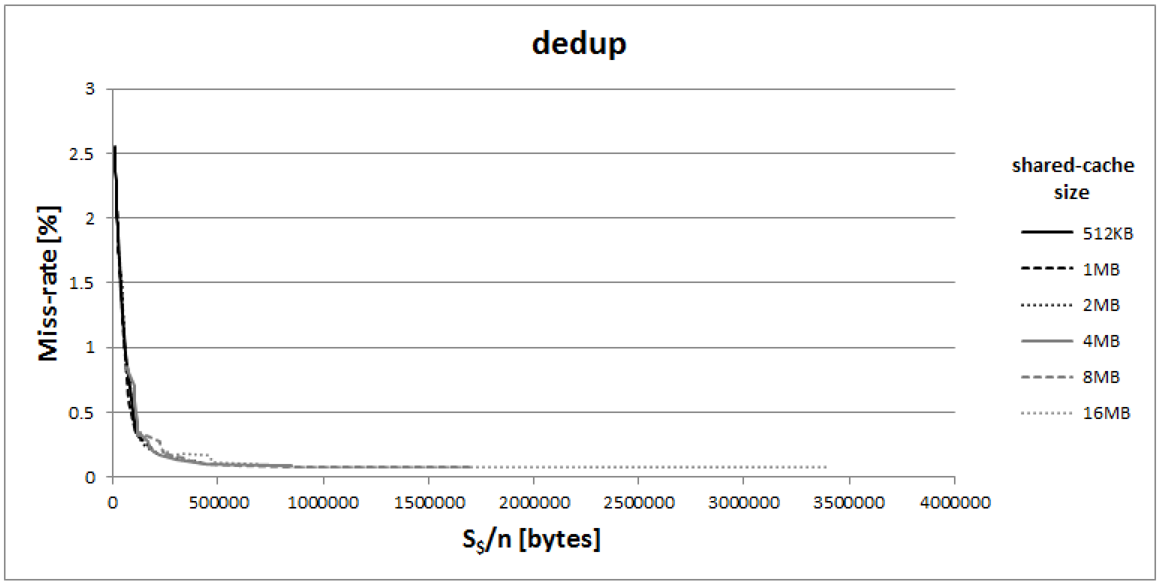


Figure 5-14: Miss-rate from simulation – dedup

Figure 5-15 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The fitted curve is quite close to the actual data-point – a difference of up to ~0.2% around 250KB on the horizontal axis.

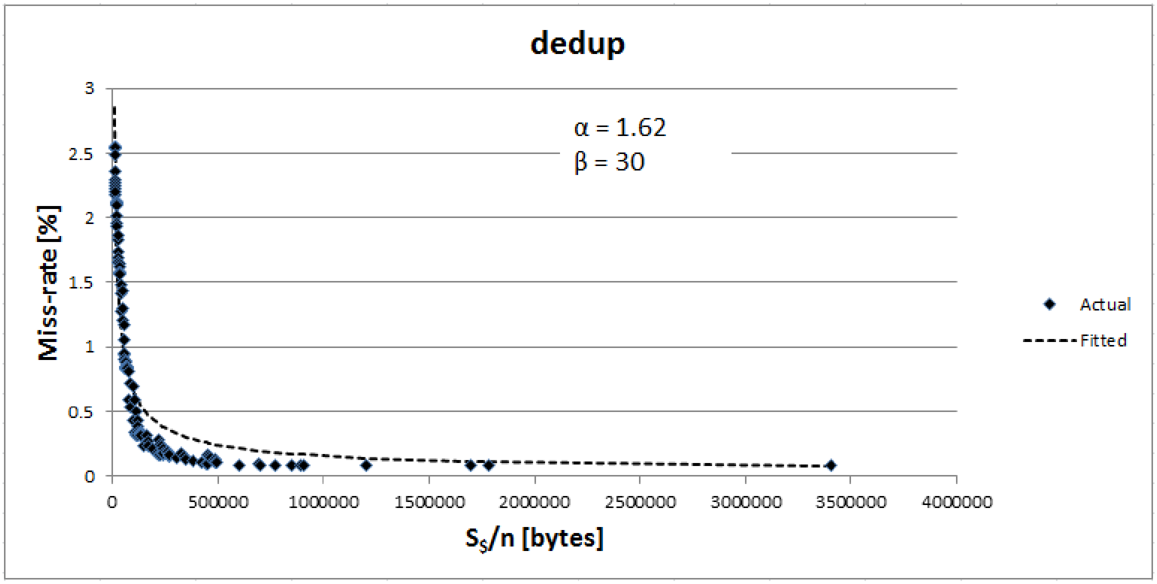


Figure 5-15: Miss-rate model fitting – dedup

Figure 5-16 shows the comparison of the actual performance and the performance that is predicted by the analytical model. The performance curves are very close, because the fitted miss-rate curve is very close to the actual miss-rate data-points.

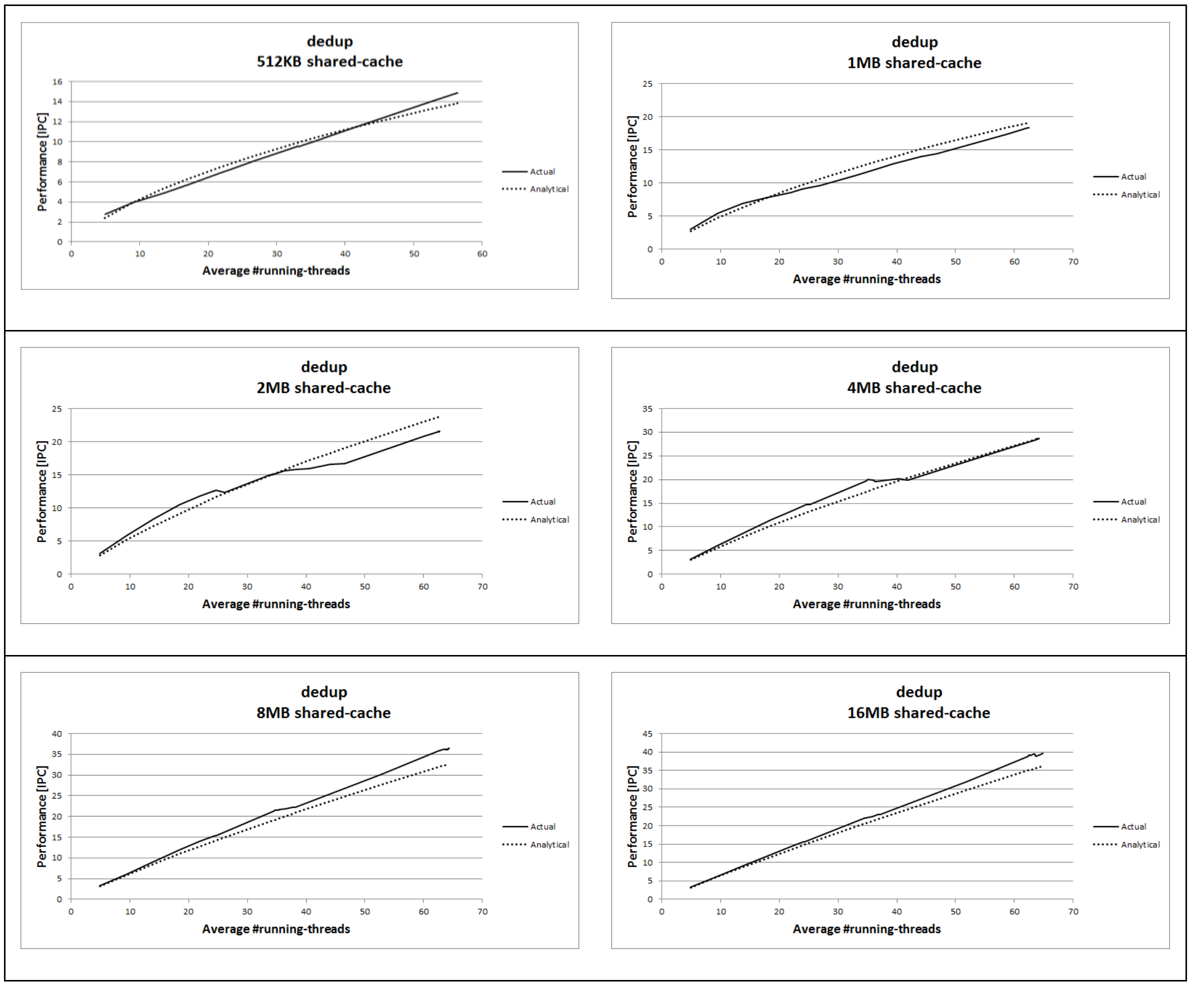


Figure 5-16: Actual vs. analytical performance with different cache sizes - dedup

5.3.5. facesim

Figure 5-17 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are mostly not overlapping, indicating that for this benchmark the miss-rate is sensitive to the total cache size, not only to EPTCS.

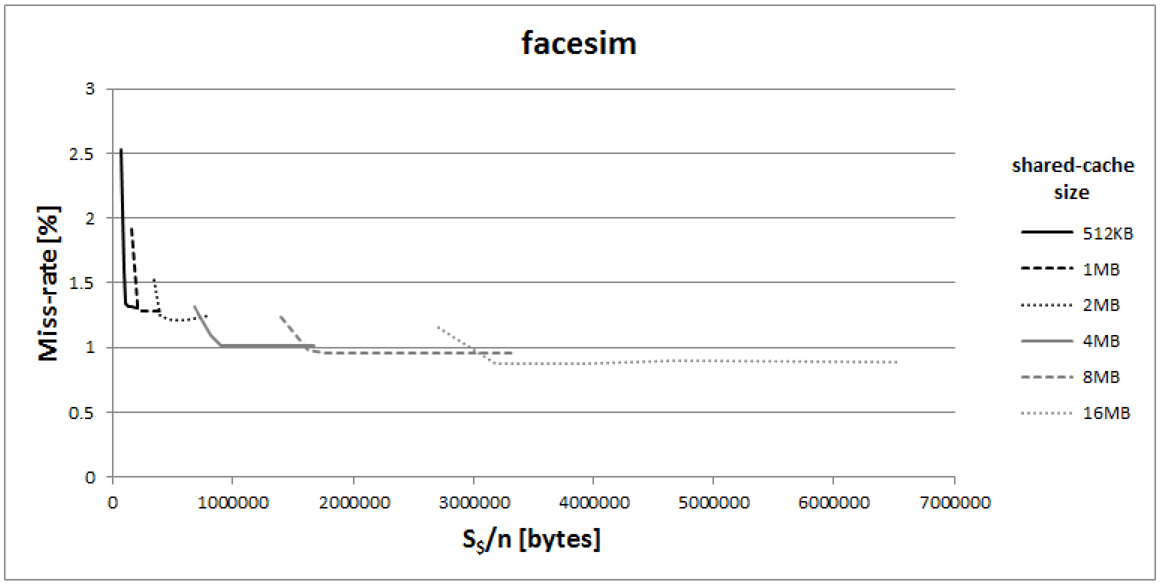


Figure 5-17: Miss-rate from simulation – facesim

Figure 5-18 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The fitted curve is quite close to the actual data-point – a difference of up to ~0.7% in the lowest region of the horizontal axis and no more than ~0.2% otherwise.

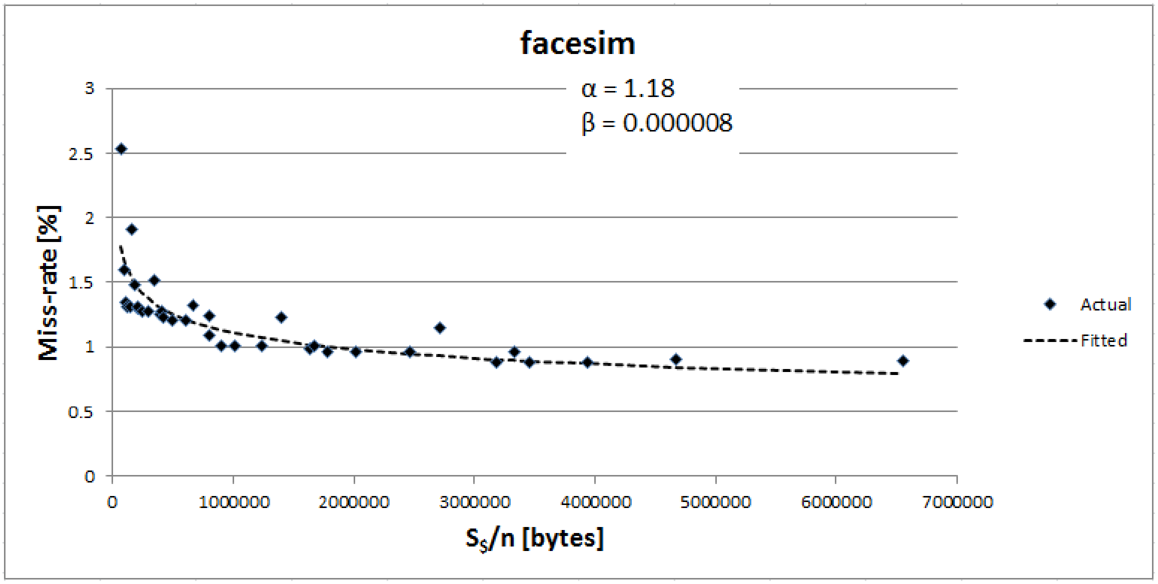


Figure 5-18: Miss-rate model fitting – facesim

Figure 5-19 shows the comparison of the actual performance and the performance that is predicted by the analytical model. The performance curves are very close, because the fitted miss-rate curve is very close to the actual miss-rate data-points, except at the highest region of the horizontal axis, which corresponds to the lowest region of the horizontal axis of Figure 5-18, where the difference between the actual and fitted miss-rate is the greatest.

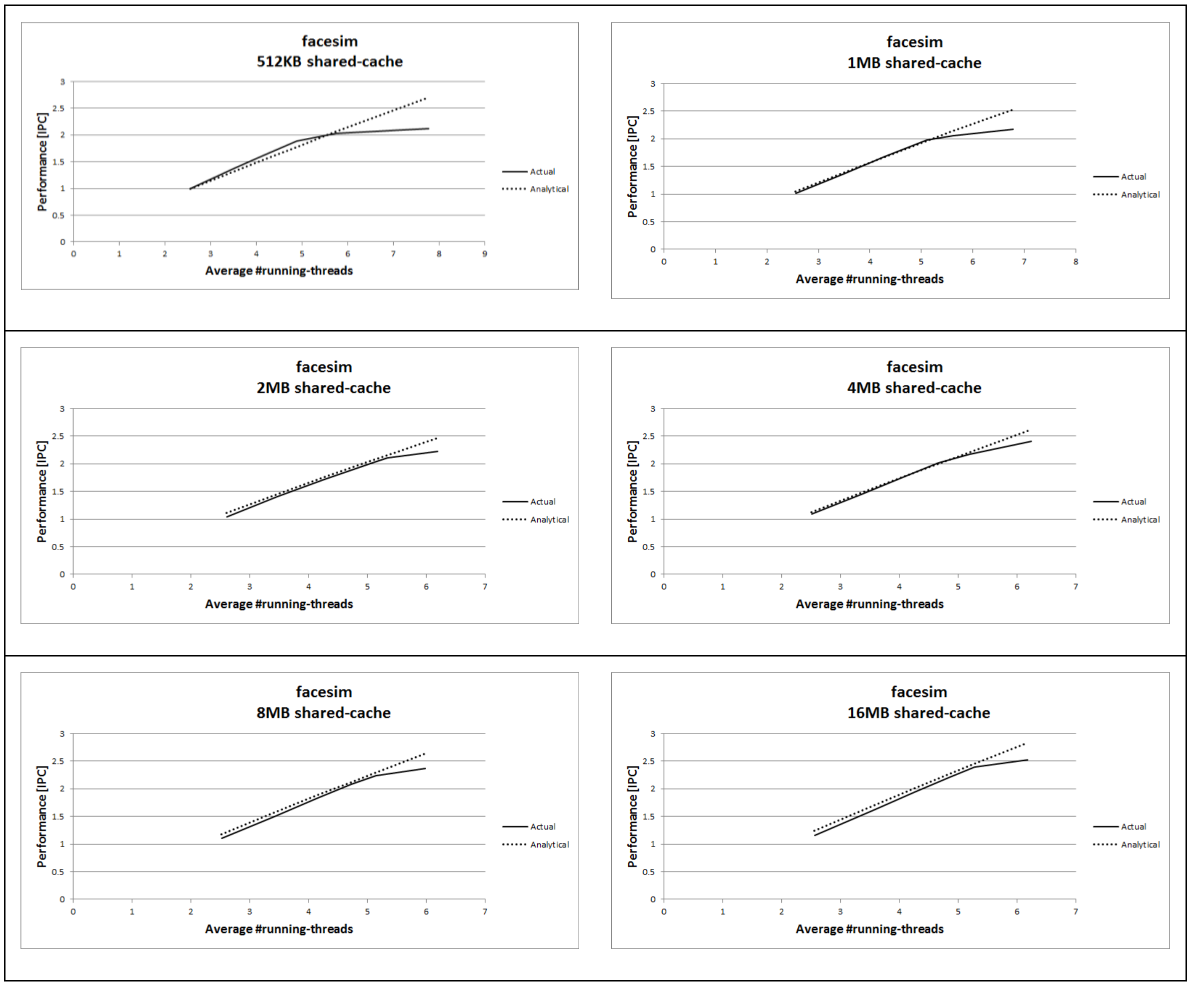


Figure 5-19: Actual vs. analytical performance with different cache sizes - facesim

5.3.6. ferret

Figure 5-20 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are mostly overlapping where applicable (i.e., where there is overlapping on the horizontal axis) and otherwise contiguous, indicating that for this benchmark the miss-rate is not sensitive to the total cache size, only to the EPTCS.

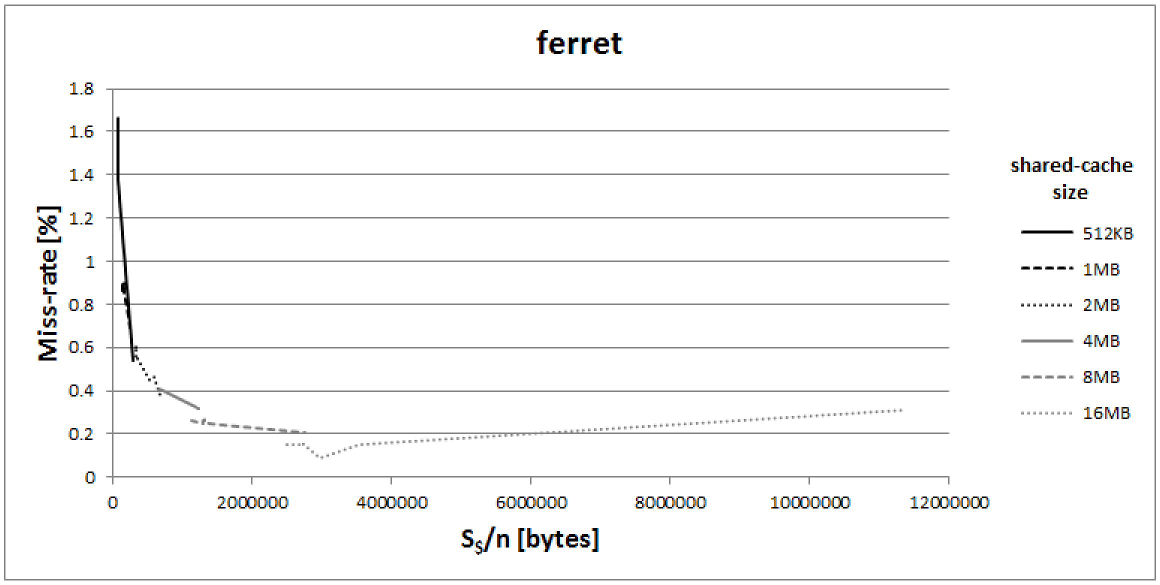


Figure 5-20: Miss-rate from simulation – ferret

Figure 5-21 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The fitted curve is quite close to the actual data-point. The rightmost data-point is visibly distant from the fitted curve but this distance is only 0.2%.

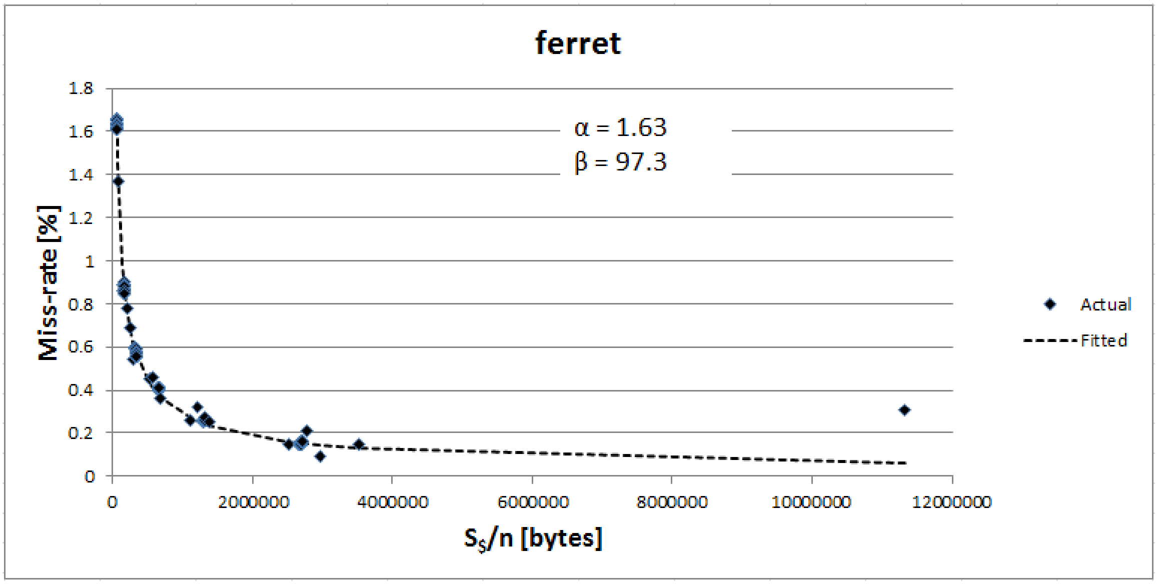


Figure 5-21: Miss-rate model fitting – ferret

Figure 5-22 shows the comparison of the actual performance and the performance that is predicted by the analytical model. The performance curves are very close, because the fitted miss-rate curve is very close to the actual miss-rate data-points.

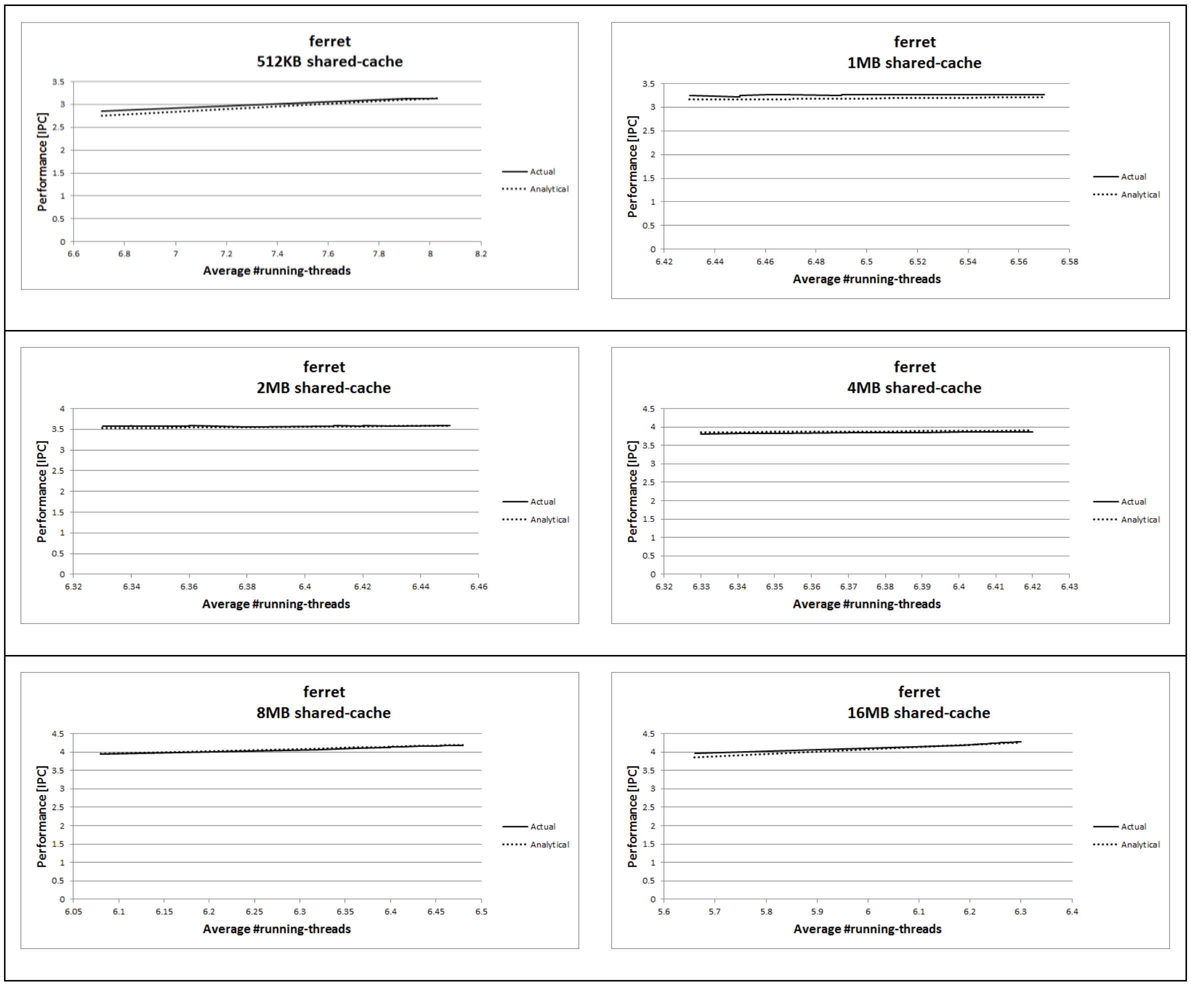


Figure 5-22: Actual vs. analytical performance with different cache sizes - ferret

5.3.7. fluidanimate

Figure 5-23 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

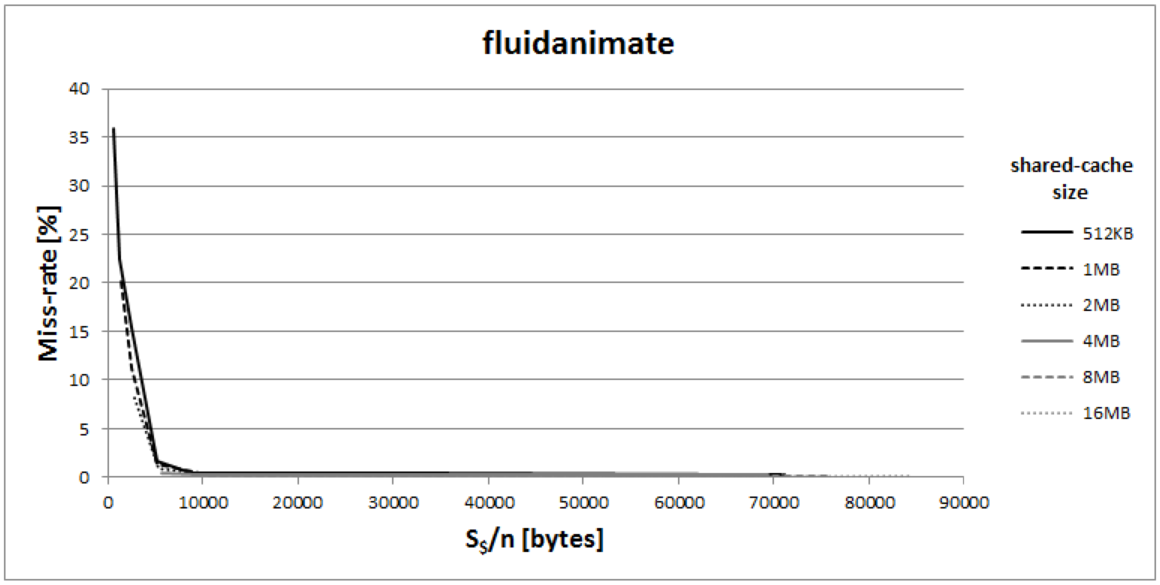


Figure 5-23: Miss-rate from simulation – fluidanimate

Figure 5-24 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The fitted curve is very close to the data-points. The largest difference of ~2.5% is in the region of 5KB on the horizontal axis.

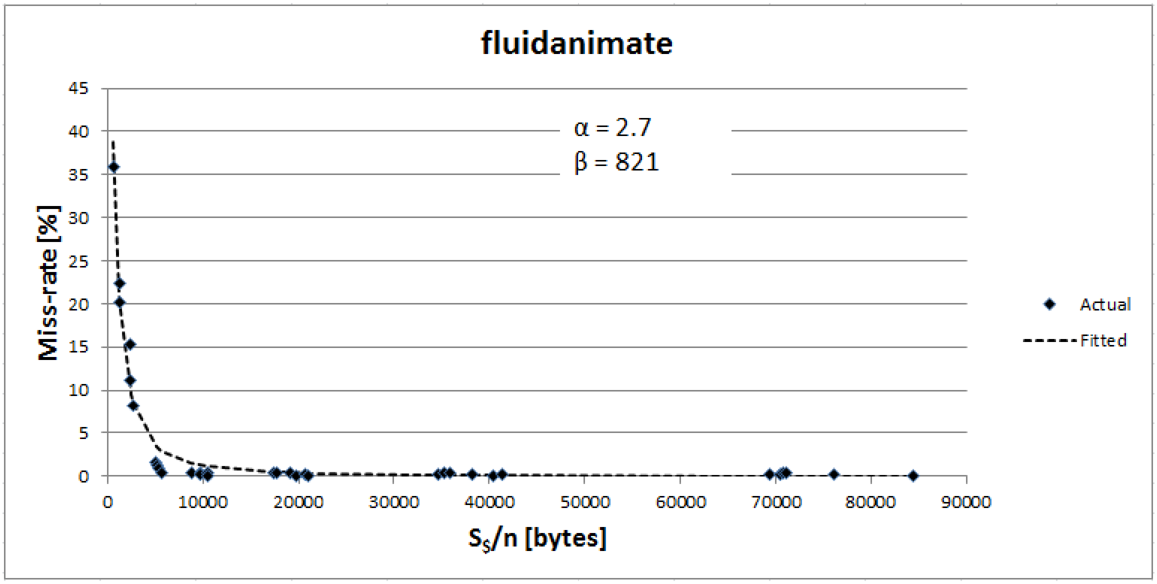


Figure 5-24: Miss-rate model fitting – fluidanimate

Figure 5-25 shows the comparison of the actual performance and the performance that is predicted by the analytical model. Although the fitted curve is very close to most of the actual data-points, in the regions with the largest difference there is a large difference between the actual and predicted performance, up to 2x. Where the predicted performance shows a valley, the actual performance has a valley, but a deeper one. This may be explained by the performance sensitivity to miss-rate variation. As can be seen, the largest difference is around EPTCS of ~5KB. Indeed this is where there is a data-point with very low miss-rate, i.e., high sensitivity and a relatively big difference between the actual and fitted miss-rates.

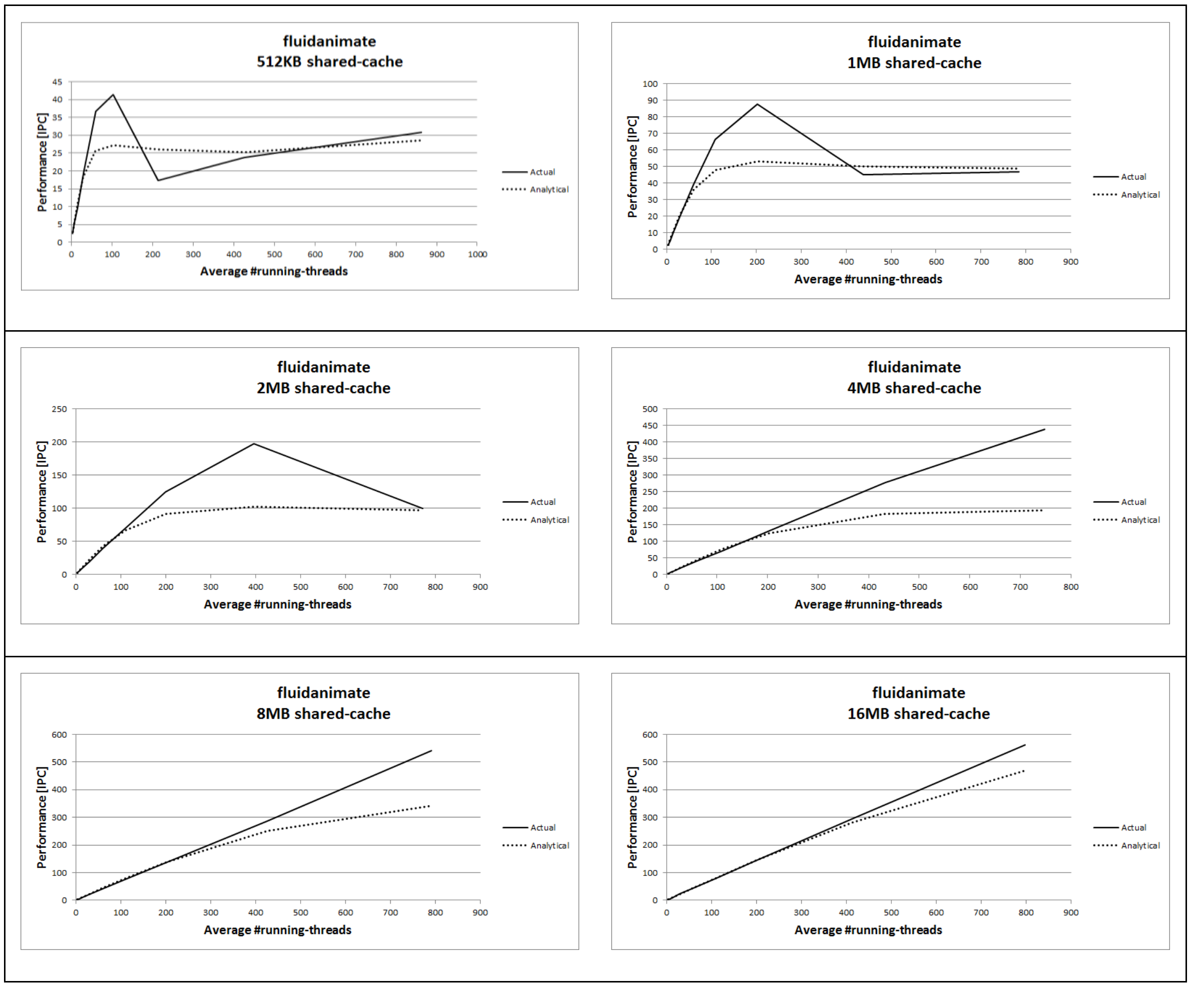


Figure 5-25: Actual vs. analytical performance with different cache sizes - fluidanimate

5.3.8. raytrace

Figure 5-26 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are overlapping except in the lower region of the horizontal axis. This indicates that for this benchmark the miss-rate is sensitive to the total cache size, not only to EPTCS.

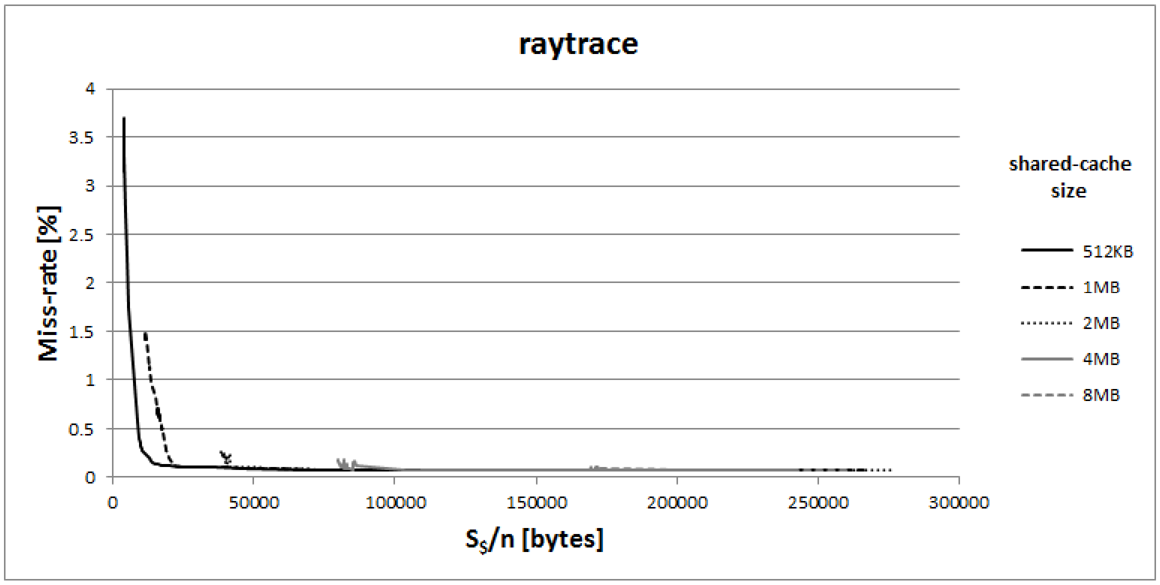


Figure 5-26: Miss-rate from simulation – raytrace

Figure 5-27 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. It shows that while the general shape is somewhat similar, the differences are relatively large in the region of 10-20KB of the horizontal axis – up to 0.7%.

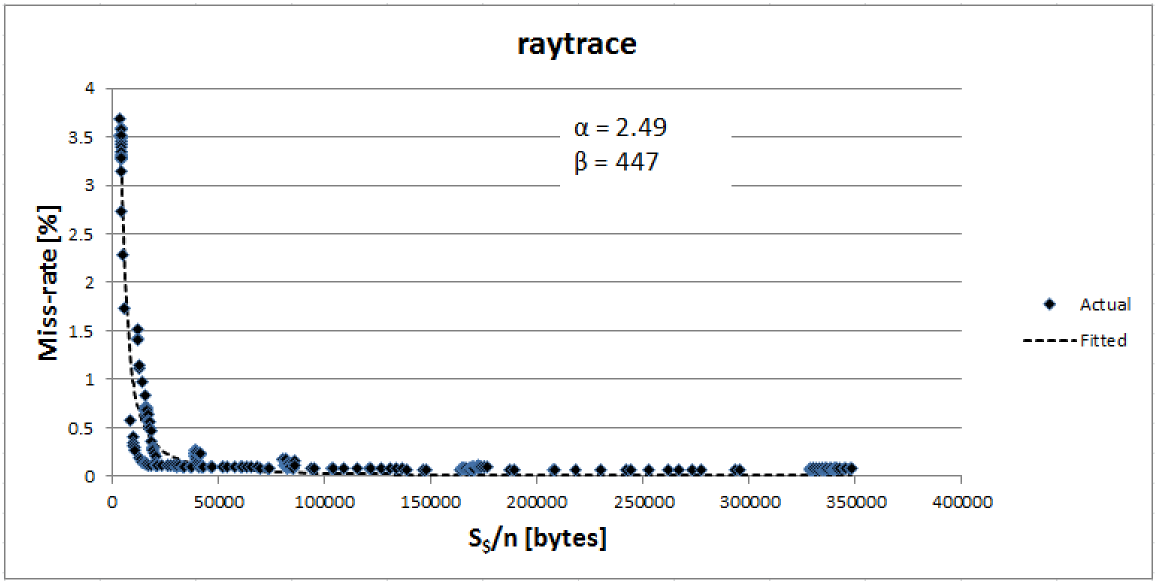


Figure 5-27: Miss-rate model fitting – raytrace

Figure 5-28 shows the comparison of the actual performance and the performance that is predicted by the analytical model. The predicted performance is very close to the actual performance, except in the region of the largest difference between actual and fitted miss-rate, where it still relatively close -- up to ~1.3x.

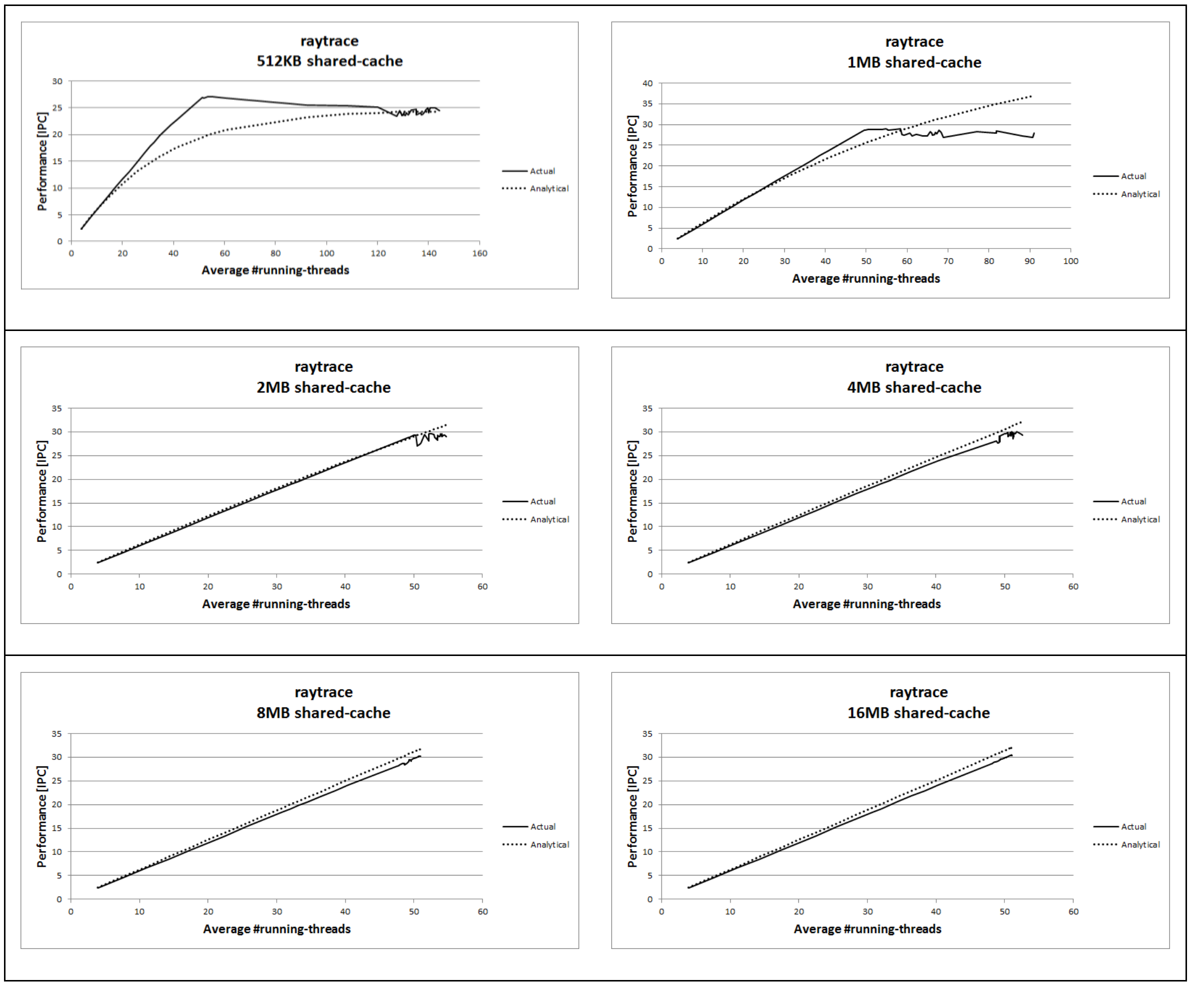


Figure 5-28: Actual vs. analytical performance with different cache sizes - raytrace

5.3.9. streamcluster

Figure 5-29 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

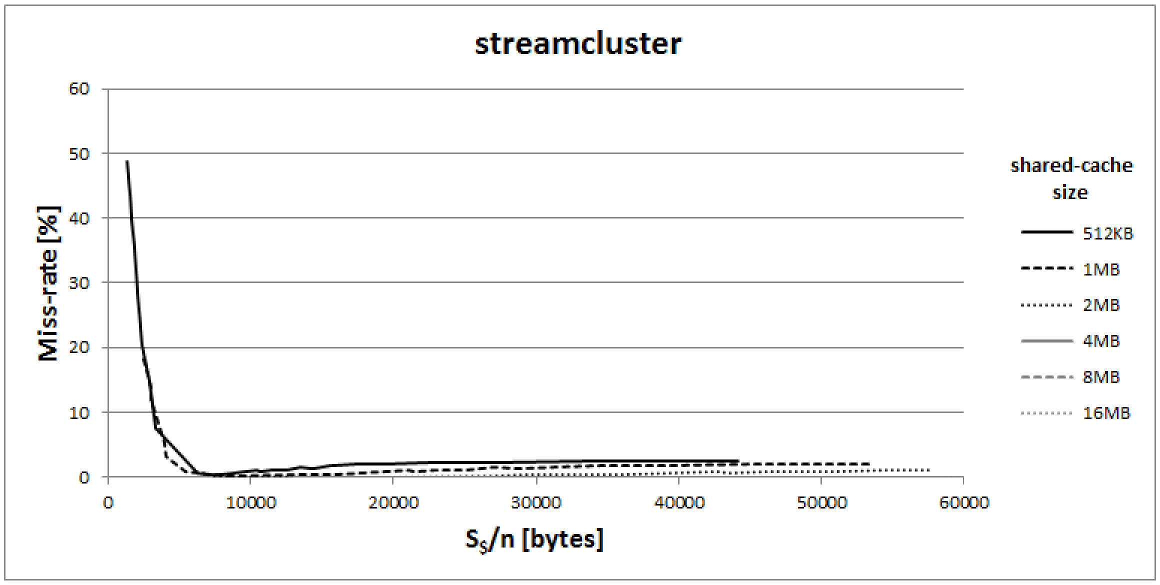


Figure 5-29: Miss-rate from simulation – streamcluster

Figure 5-30 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The fitted curve is very close to the data-points. The largest difference of ~5% is in the region of 4KB on the horizontal axis.

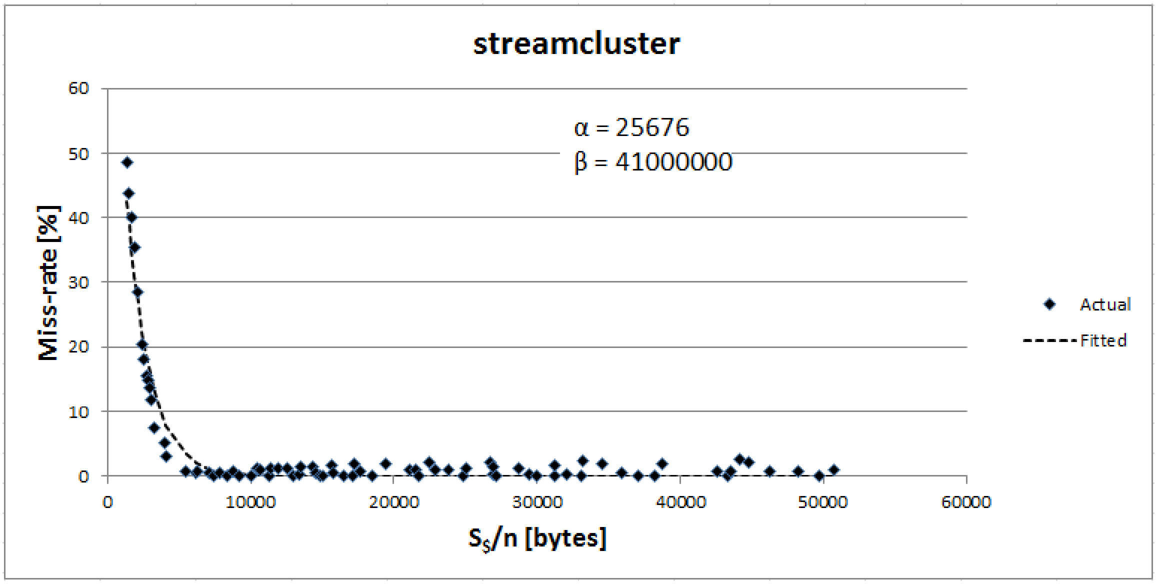


Figure 5-30: Miss-rate model fitting – streamcluster

Figure 5-31 shows the comparison of the actual performance and the performance that is predicted by the analytical model. For cache sizes of 4MB and up, the entire data-set fits in the cache so the miss-rate is zero. The corresponding portion of the fitted miss-rate is effectively zero[11] and therefore the actual and analytical performance curves overlap. For cache sizes of 2MB and below, although the analytical performance curve shows the same general trend as the actual performance curve, the actual performance is as high as 2x of the analytical performance with cache of 1MB and 200 threads, i.e., at EPTCS of ~5KB. As shown in Figure 5-30, this is where the actual miss-rate becomes very small (~0.8%), i.e., the performance sensitivity to the miss-rate becomes large, and the fitted miss-rate is significantly larger (~3.3%).

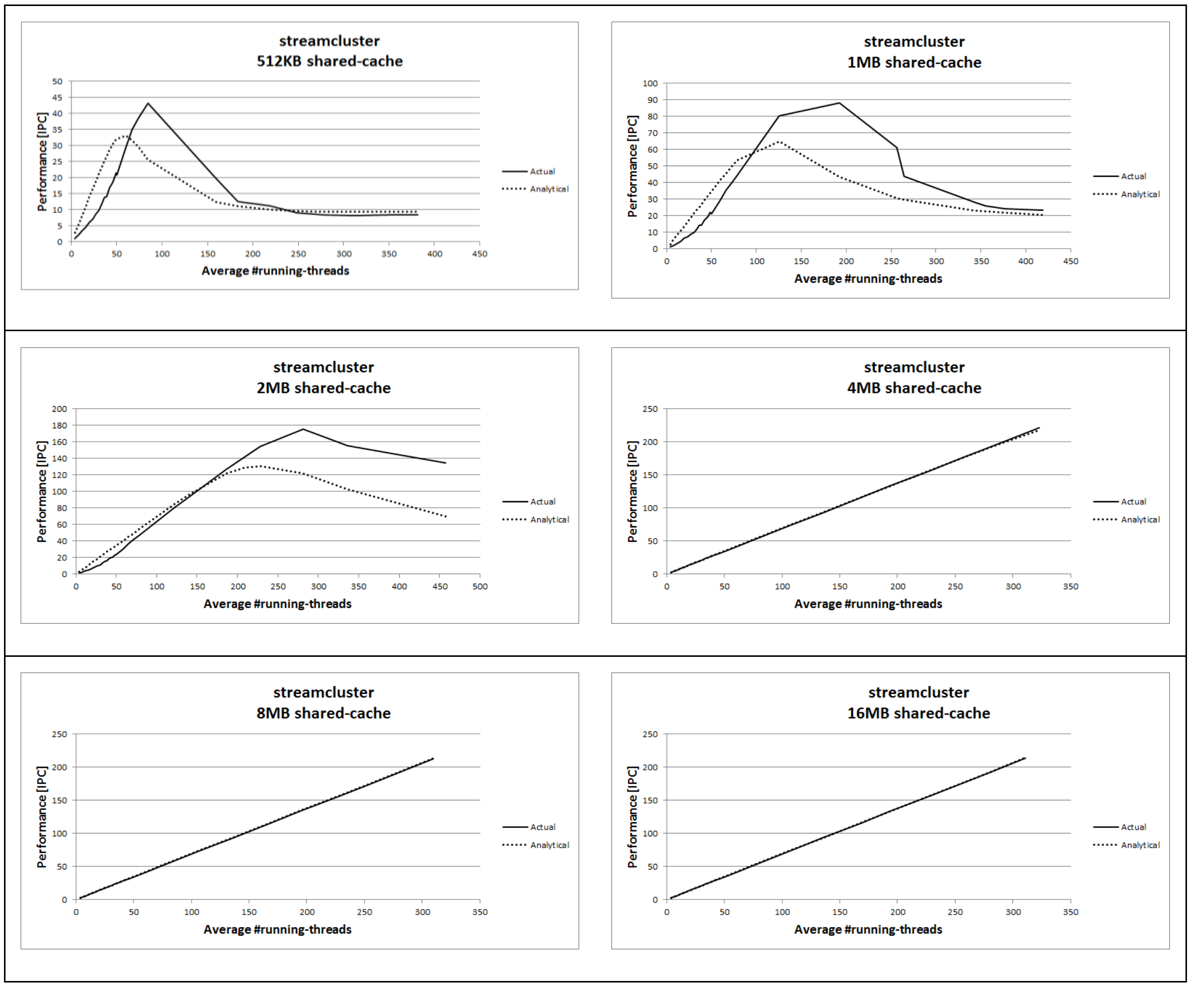


Figure 5-31: Actual vs. analytical performance with different cache sizes - streamcluster

[11] The fitted miss-rate can never have a value of exactly 0 because of the way formula (5-2) is defined but can be very small, which is the case here.

5.3.10. swaptions

Figure 5-32 shows the miss-rate as a function of EPTCS, for each cache size separately. The graphs are mostly not overlapping, indicating that for this benchmark the miss-rate is sensitive to the total cache size, not only to EPTCS.

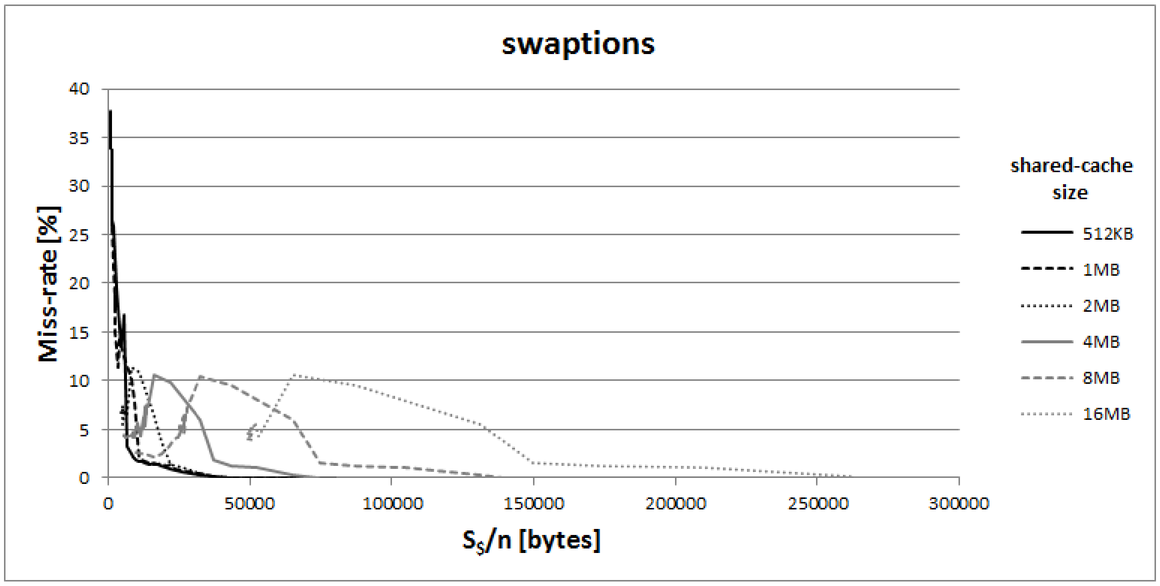


Figure 5-32: Miss-rate from simulation – swaptions

Figure 5-33 shows the actual miss-rate data-points and the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. It shows that while the general shape is somewhat similar, the differences are quite large, in particular since there are multiple data points with the same EPTCS but significantly different miss-rate, e.g., in EPTCS of 33KB there are data points with miss-rate difference of 10%.

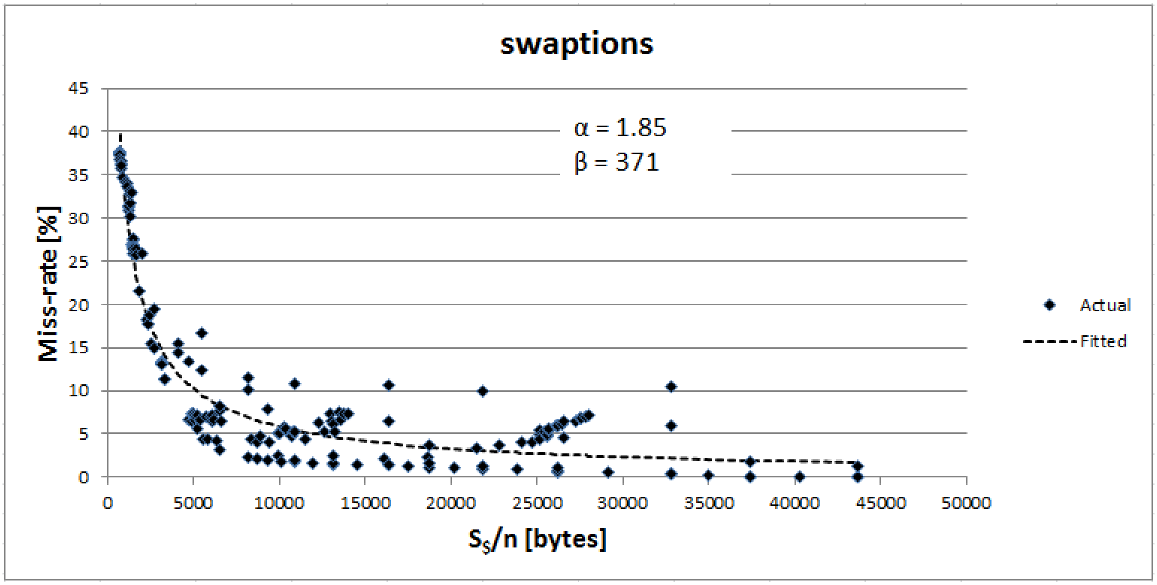


Figure 5-33: Miss-rate model fitting – swaptions

Figure 5-34 shows the comparison of the actual performance and the performance that is predicted by the analytical model in the left column and the respective actual performance vs. the number of worker threads in the right column. Although the predicted performance is monotonously increasing, the actual performance has a valley for all cache sizes. Note that this benchmark is embarrassingly parallel but the worker threads contend on the dynamic memory allocation heap (see section 4.3.10).

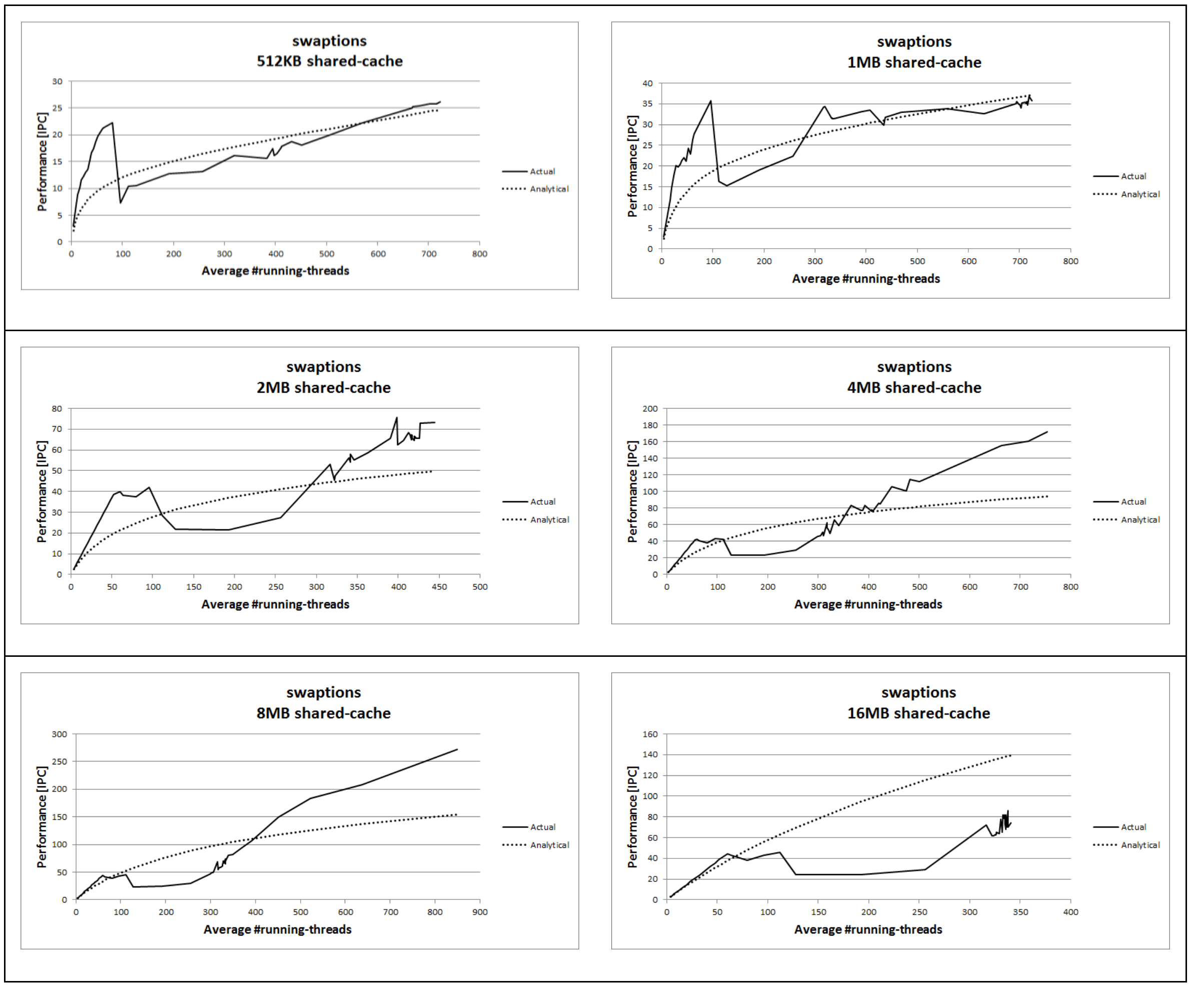


Figure 5-34: Actual vs. analytical performance with different cache sizes - swaptions

5.3.11. vips

Figure 5-35 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

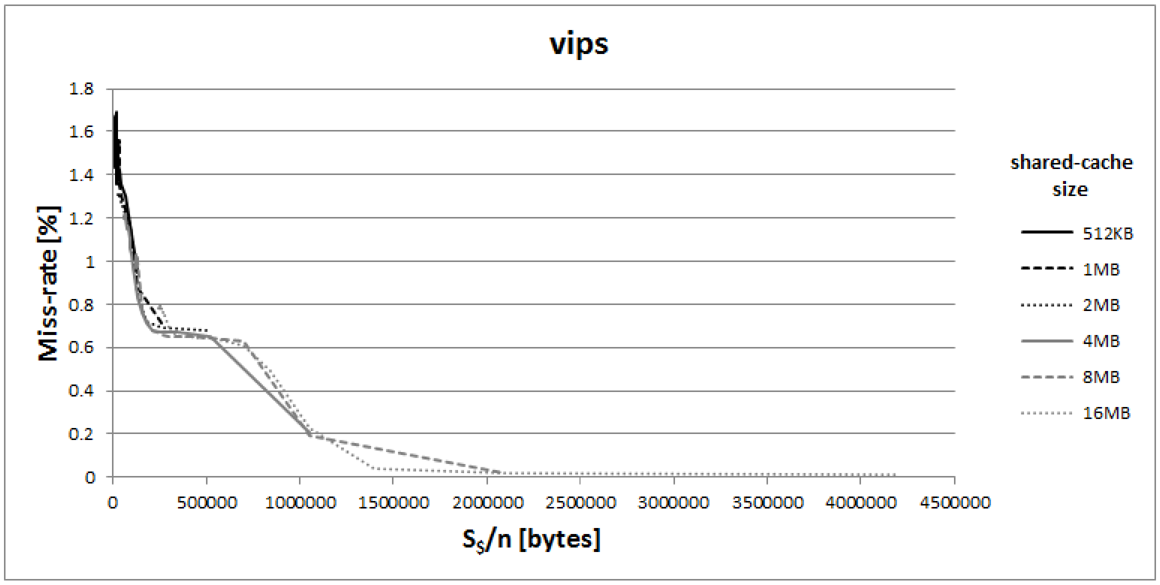


Figure 5-35: Miss-rate from simulation – vips

Figure 5-36 shows all the actual miss-rate data-points, for all cache sizes and all thread counts as measured by the simulation. It also shows the graph of the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The actual miss-rate is small (<1.8%) across all data-points and therefore the difference between actual and fitted miss-rate is small. The difference is relatively large with higher EPTCS but this corresponds to lower numbers of running threads. As indicated in section 5.2 the performance sensitivity to miss-rate is proportional to the number of running threads and therefore the effect of the miss-rate difference with smaller number of running threads should be small.

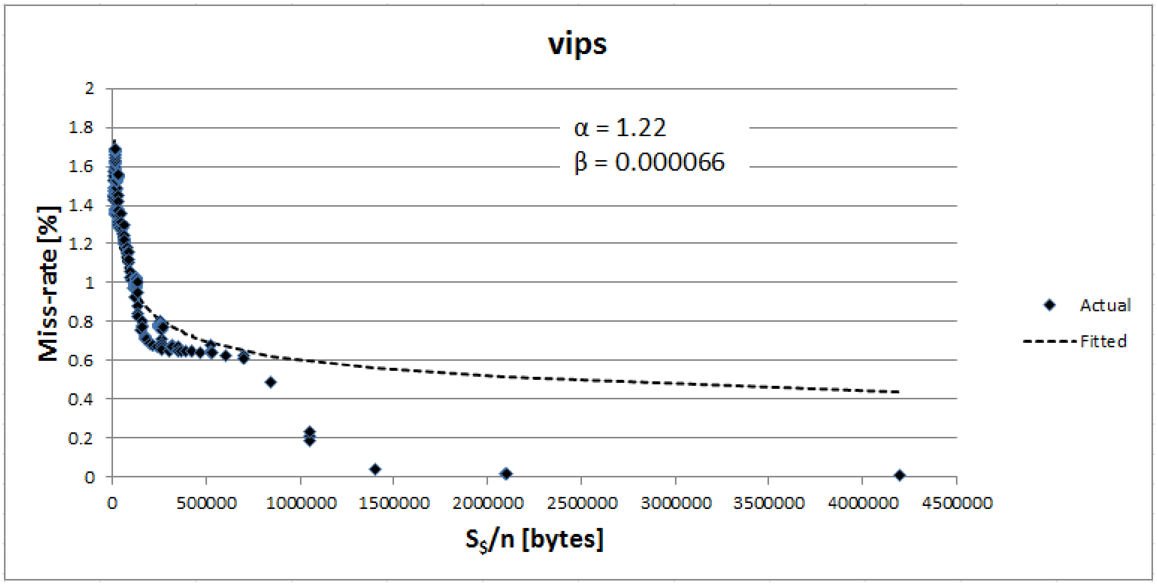


Figure 5-36: Miss-rate model fitting – vips

Figure 5-37 shows the comparison of the actual performance and the performance that is predicted by the analytical model. Since the difference between the fitted and actual miss-rate is small, the actual and predicted performance are very close.

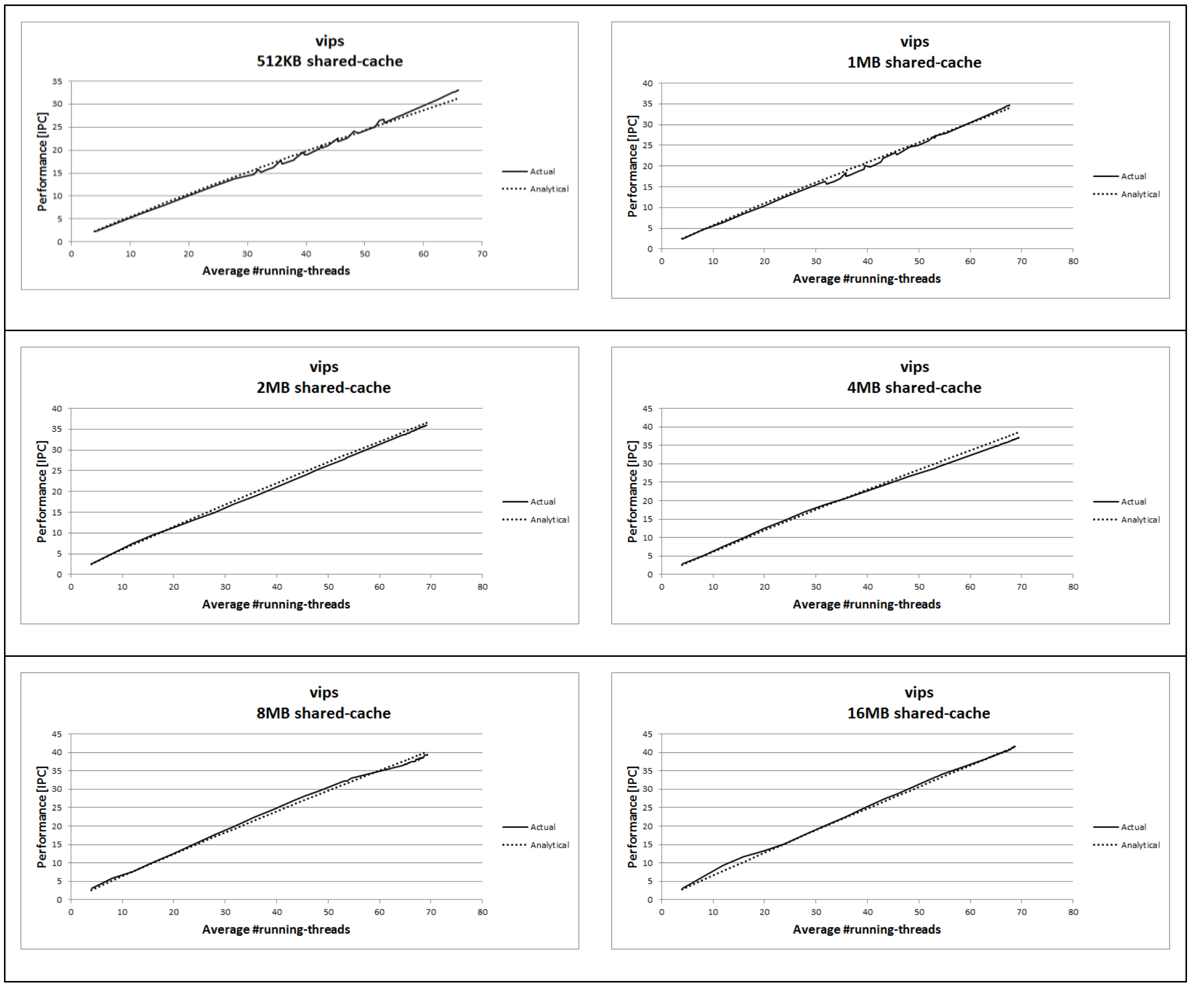


Figure 5-37: Actual vs. analytical performance with different cache sizes - vips

5.3.12. x264

Figure 5-38 shows the miss-rate as a function of EPTCS, for each cache size separately. The overlapping of the graphs indicates that for this benchmark the miss-rate is not sensitive to the total cache size, only to EPTCS.

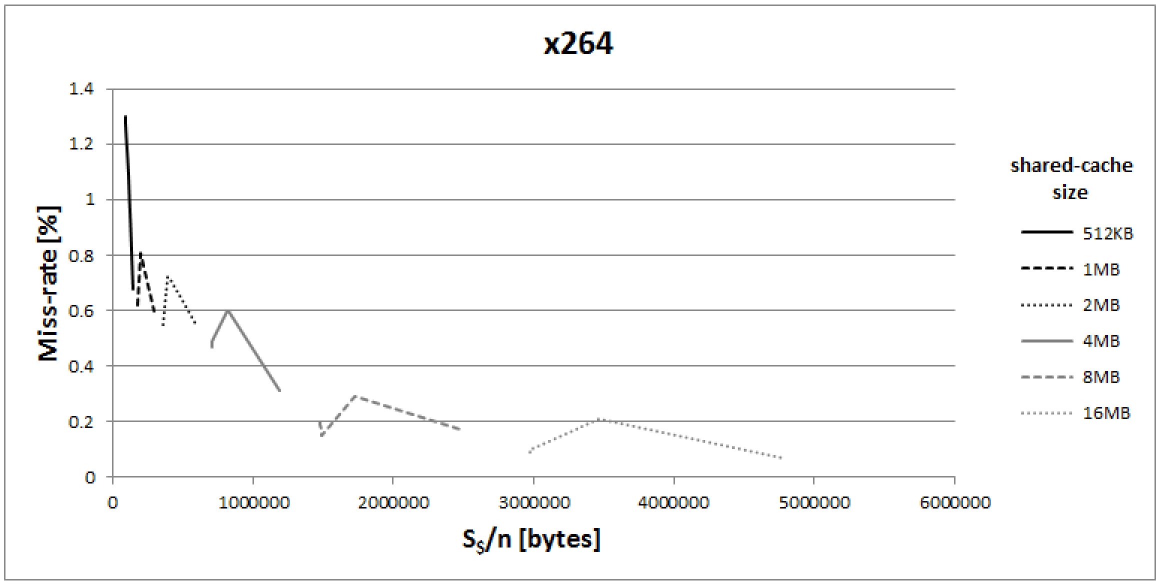


Figure 5-38: Miss-rate from simulation – x264

Figure 5-39 shows all the actual miss-rate data-points, for all cache sizes and all thread counts as measured by the simulation. It also shows the graph of the fitted closed form of miss-rate model formula (5-2), with α and β that determine this closed form. The actual miss-rate is small (<1.4%) across all data-points and therefore the difference between actual and fitted miss-rate is small. The difference is relatively large with higher EPTCS but this corresponds to lower numbers of running threads. As indicated in section 5.2 the performance sensitivity to miss-rate is proportional to the number of running threads and therefore the effect of the miss-rate difference with smaller number of running threads should be small.

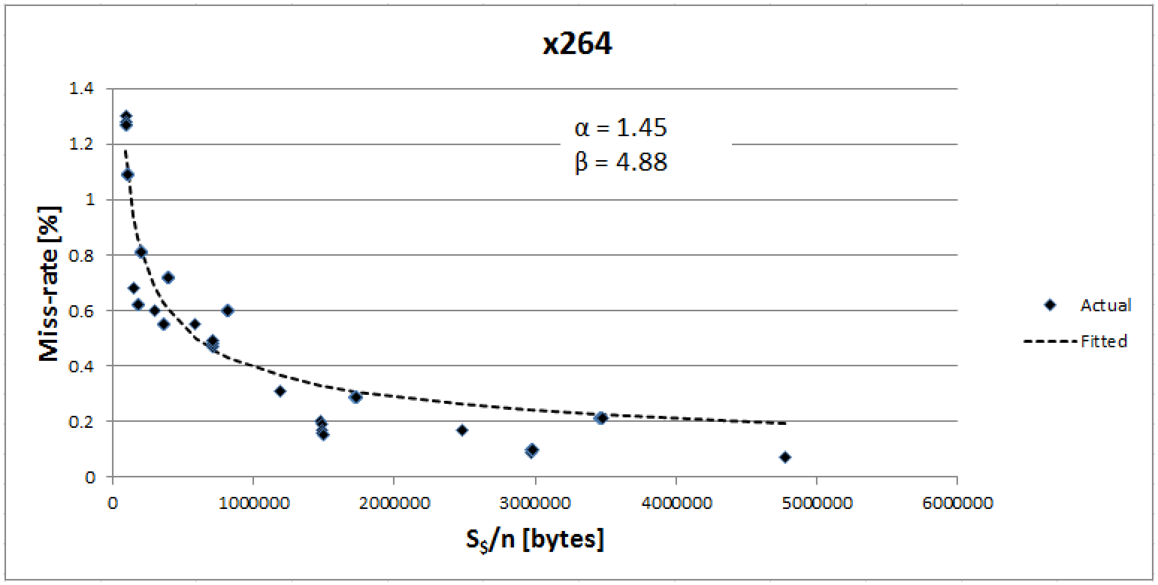


Figure 5-39: Miss-rate model fitting – x264

Figure 5-40 shows the comparison of the actual performance and the performance that is predicted by the analytical model. Since the difference between the fitted and actual miss-rate is small, the actual and predicted performance are very close.

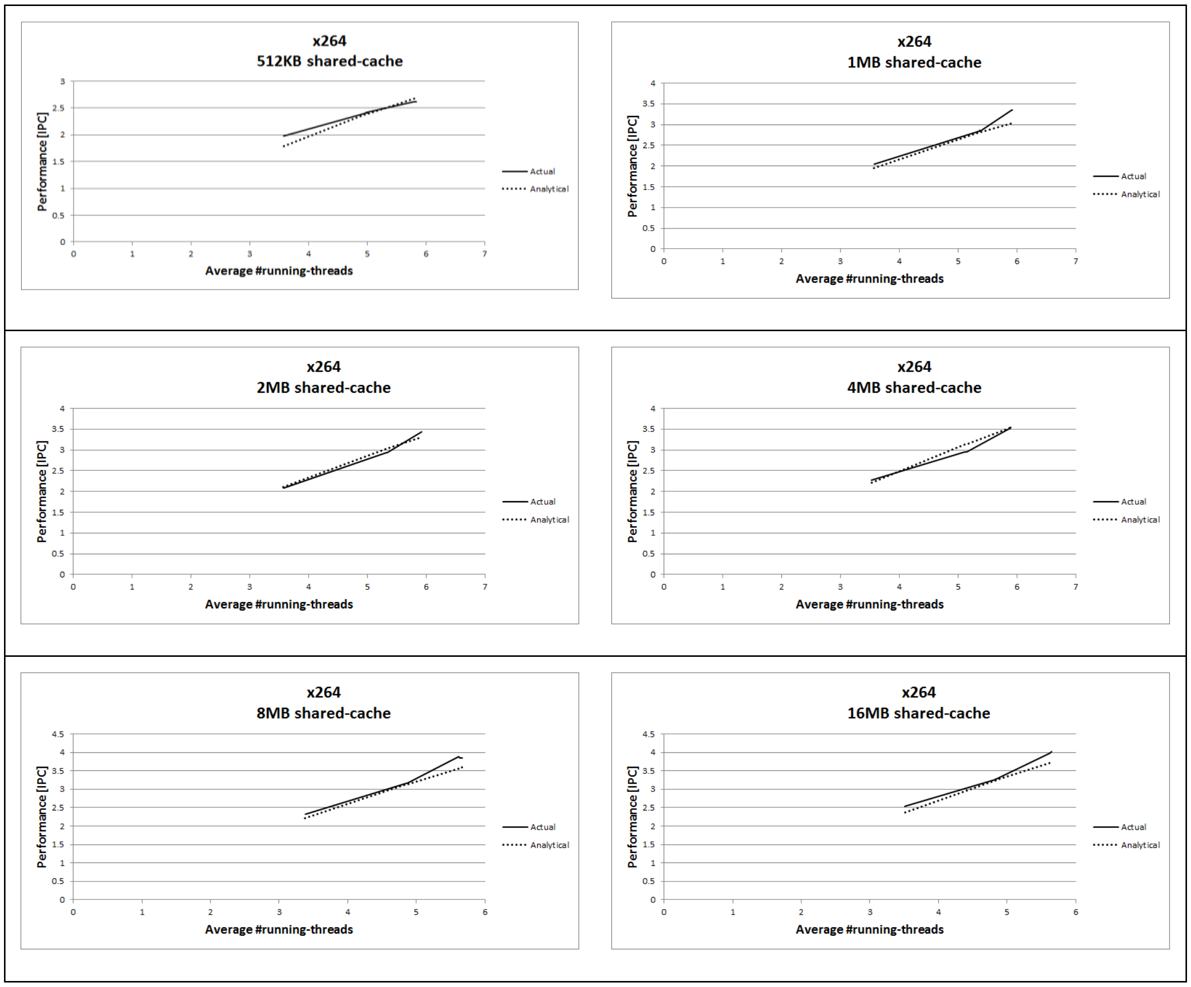


Figure 5-40: Actual vs. analytical performance with different cache sizes – x264

5.4. Conclusions

We study the applicability of the cache performance (miss-rate) analytical model depicted in formula (5-2) that was proposed in the literature. We measure the actual cache performance on the diverse workloads of the Parsec benchmark suite and different cache sizes, from which we extract the workload-dependent parameters of the analytical model using curve-fitting. With these parameters we get a closed-form of the cache performance analytical model, which we then compare to the actual cache performance.

Similarly, we derive a closed-form of the analytical performance model that is depicted in formula (2-5) by placing the closed-form analytical cache performance model in it. We compare the performance predicted by the analytical model to the actual performance.

We make the following conclusions from our cache analytical modeling study:

1. Formula (5-2) is a good *first order* analytical model for cache performance under parallel workloads
2. The sensitivity of the analytical performance model of formula (2-5) to the miss-rate makes Formula (5-2) inadequate for analytical performance prediction.

Chapter 6. Summary

In this research we developed a simulator for a simple many-core architecture model and used it to study some aspects of the behavior of the Parsec benchmark suite on that architecture model – the parallelism scalability and shared-cache behavior.

Our simulator can simulate hundreds and thousands of cores and run any Linux program without re-compilation. This implies that it does not require special development toolset (compiler, linker, libraries), no source code modifications and can simulate programs written in any language (by simulating the execution environment program such as the JVM for Java programs). The simulator maintains the effect of inter-thread communication so that the timing effect on the program is preserved. In particular, when a thread blocks waiting for some other computation to complete, this is correctly simulated, hence the simulation captures the algorithmic parallelism effects on the performance. The simulator takes the architecture parameters (such as number of cores, number of thread contexts, cache size, memory hierarchy access latencies etc.) as invocation parameters, facilitating the study of each parameter's effect on the workload.

We studied the parallelism degree limitations, what we call *parallelism scalability*, of the benchmarks in the Parsec benchmarks suite. This characterizes a program inherent parallelism limitation by running it on a perfectly parallel architecture (one with no parallelism limiting factors, i.e., no shared resource) with perfect memory hierarchy (1 cycle latency for all memory accesses). On such architecture, any deviation from perfect scaling of the performance with the architecture is necessarily due to an algorithm in the program, e.g. a serial portion that is a performance bottleneck. We find that most benchmarks achieve peak performance with 128 threads or less, with one as low as 4 threads and another as low as 8 threads. This implies that these benchmarks are not suitable for exploring architectures with higher degree of parallelism.

Another aspect of the Parsec benchmark suite that we studied is shared-cache performance (miss-rate). Specifically, we compared it against an analytical model that is proposed in the literature. We find that for most benchmarks the actual cache performance is compatible with the analytical model, but for some it is not. In particular, the model assumes that the cache performance depends only on the quotient of the cache size and the number of threads rather than on each individually. Simulation results show that this is not always the case. There are benchmarks that even when the cache performance depends only on the quotient, it still deviates significantly from the analytical model. Finally, even benchmarks whose cache performance depends only on the quotient and is compatible with the analytical model, when put into the analytical performance model there is a big difference from the actual performance, because the performance is increasingly sensitive to differences in miss-rate as the miss-rate decreases. Thus, in small miss-rates, small differences between the actual and analytical miss-rate translates to big differences between actual and analytical performance.

Appendix: simulation environment

Hardware platform: HP Proliant DL785 G5, with 8 AMD Opteron™ Processor 8356 Quad-Core (total of 32 cores), 128GB RAM

Operating System: Linux Ubuntu 12.04 Server, 64-bit

Pin binary instrumentation framework: version 2.12

Parsec benchmark suite: version 2.1, with the following patches:

1. Syntax error in open()
2. Deadlock in ferret
3. Missing barrier in streamcluster

Compiled with GCC version 4.6.3 in **gcc-hooks** configuration mode

Simulator Pintool: compiled with GCC version 4.6.3

References

[1] G.M. Amdahl, “Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities,” Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp. 483-485.

[2] C. Bienia, “Benchmarking Modern Multiprocessors”, Ph.D. Thesis. Princeton University, January 2011.

[3] C. Bienia and K. Li, “PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors”, In Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009.

[4] Z. Guz et al, "Threads vs. Caches: modeling the behavior of parallel workloads", In ICCD, October 2010.

[5] Flynn, Michael J., and Albert Podvin. "Shared resource multiprocessing." Computer 5.2 (1972): 20-28.

[6] Magnusson, Peter S., et al. "Simics: A full system simulation platform." Computer 35.2 (2002): 50-58.

[7] Bedichek, Robert. "SimNow: Fast platform simulation purely in software." Hot Chips. Vol. 16. 2004.

[8] Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH Computer Architecture News 39.2 (2011): 1-7.

[9] Manavski, Svetlin A., and Giorgio Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment." BMC bioinformatics 9.Suppl 2 (2008): S10.

[10] Tarditi, David, Sidd Puri, and Jose Oglesby. "Accelerator: using data parallelism to program GPUs for general-purpose uses." ACM SIGARCH Computer Architecture News. Vol. 34. No. 5. ACM, 2006.

[11] Keramidas, Georgios, Nikolaos Strikos, and Stefanos Kaxiras. "Multicore Cache Simulations Using Heterogeneous Computing on General Purpose and Graphics

Processors." Digital System Design (DSD), 2011 14th Euromicro Conference on. IEEE, 2011.

[12] J. E. Miller et al, “Graphite: A distributed parallel simulator for multicores”, HPCA-16, January 2010.

[13] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." ACM SIGPLAN Notices. Vol. 40. No. 6. ACM, 2005.

[14] Stallings, William. Operating systems: internals and design principles. Prentice Hall, 2008.

[15] Adve, Sarita V., and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial." computer 29.12 (1996): 66-76.

[16] Gustafson, John L. "Reevaluating Amdahl's law." Communications of the ACM 31.5 (1988): 532-533.

[17] Lewis, Bil, and Daniel J. Berg. Multithreaded programming with Pthreads. Vol. 2550. Sun Microsystems Press, 1998.

[18] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[19] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, “An analytical model for designing memory hierarchies” IEEE Transactions on Computers, vol. 45, no 10, October 1996

[20] Chow, C. K. "Determination of cache's capacity and its matching storage hierarchy." Computers, IEEE Transactions on 100.2 (1976): 157-164.

[21] Madsen, Kaj, Hans Bruun, and Ole Tingleff. "Methods for non-linear least squares problems." (1999).

[22] Joachim Wuttke: lmfit - a C/C++ routine for Levenberg-Marquardt minimization with wrapper for least-squares curve fitting, based on work by B. S. Garbow, K. E. Hillstrom, J. J. Moré, and S. Moshier. Version 3.3, retrieved on March 9th, 2013 from http://joachimwuttke.de/lmfit/.

ההשפעה ההדדית בין תוכנות לרכיבי הארכיטקטורה במעבדים בעלי מקביליות גבוהה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר

מגיסטר למדעים בהנדסת חשמל

עובד יצחק

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

תמוז תשע"ג   חיפה   יוני 2013

תודות

המחקר נעשה בהנחיית פרופסור אורי ויזר, פרופסור אבינועם קולודני ופרופסור עדית קידר, בפקולטה להנדסת חשמל.

תקציר

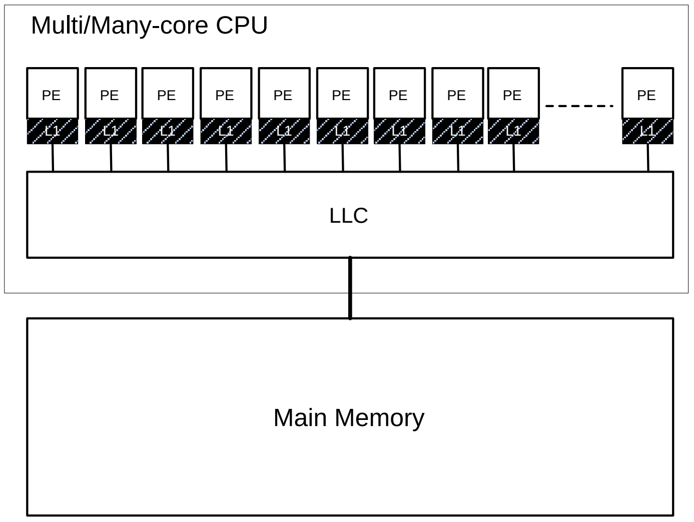
אנו משתמשים במודל פשוט של ארכיטקטורת מעבד מרובה ליבות לאיפיון ההתנהגות של תוכניות מקביליות תחת ארכיטקטורות עם מספר גדול של ליבות (מאות עד אלפים בודדים). בפרט אנו מאפיינים את התנהגות תוכניות הבדיקה המגוונות לארכיטקטורות מקביליות של חבילת Parsec ובפרט את שני ההיבטים הבאים:

- מידת היכולת של התוכנית לנצל מספר הולך וגדל של מעבדים.
- השפעת מספר הולך וגדל של מעבדים על ביצועי זיכרון מטמון משותף (miss-rate) והשוואה עם מודל אנליטי שהוצע בספרות.

לצורך איפיון התנהגות התוכניות פיתחנו סימולטור המממש את מודל המעבד מרובה הליבות. הסימולטור מבוסס binary instrumentation תחת Linux ולכן מסוגל להריץ כל תוכנית שניתן להריץ על מכונת Linux (כלומר התוכנית לא חייבת להיות בצורה של קובץ executable. למשל תוכניות Java שאינן executable ירוצו ע"י סימולציה של ה-JVM, שהוא עצמו executable).

מודל הארכיטקטורה

המודל מתואר באיור 1. המודל מורכב ממערך של יחידות עיבוד (PE – Processing Element, שבהקשרים שונים נקרא לפעמים Core או ALU) במודל זיכרון משותף עם היררכיית זיכרון של שלוש רמות: זיכרון מטמון פרטי ליחידת עיבוד (private cache, L1), זיכרון מטמון משותף לכל יחדות העיבוד (shared cache, L2) וזיכרון חיצוני.



איור 1: מודל הארכיטקטורה

מודל התיזמון הוא מודל פשוט של שיהוי קבוע: לכל הוראת מעבד יש שיהוי קבוע לחלק החישובי ושיהוי נוסף לגישה לזיכרון, אם יש כזו, שהיא בעצמה בעלת שיהוי קבוע התלוי ברמת היררכיית הזיכרון שבה נמצא הנתון. יחידת הזמן שאנו משתמשים בה היא *מחזור שעון* במקום יחידות זמן רגילות, מה שמאפשר לחסוך את הצורך בתדר העבודה כפרמטר של המודל, מבלי לאבד מהכלליות.

פרמטרי מודל הארכיטקטורה מפורטים בטבלה 1 (ההשהיה של זיכרון המטמון הפרטי L1 מובנית בפרמטרי התוכנית – מתואר בהמשך).

פרמטר	תיאור
N_{PE}	מספר יחידות העיבוד
$S_{\$}$	גודל זיכרון המטמון (המשותף L2)
N_{max}	המספר המרבי של חוטים (threads) $\leq N_{PE}$
CPI_{exe}	ההשהיה הממוצעת לחלק החישובי של הוראת מעבד
$t_{\$}$	ההשהיה של גישה לזיכרון המטמון (המשותף L2)
t_m	ההשהיה של גישה לזיכרון החיצוני

טבלה 1: פרמטרי מודל הארכיטקטורה

פרמטרי מודל התוכנית מפורטים בטבלה 2.

פרמטר	תיאור
n	מספר החוטים (threads)
r_m	החלק היחסי של הוראות מעבד הניגשות לנתונים בזיכרון מתוך סך כל ההוראות שהמעבד מבצע $[0 \leq r_m \leq 1]$
$P_{hit}(S_{\$},n)$	ביצועי זיכרון המטמון (המשותף L2) בהינתן שגודלו S$ והוא משמש n חוטים

טבלה 2: פרמטרי מודל התוכנית

מדד הביצועים שבו אנו משתמשים הוא מספר הוראות המעבד למחזור (Instructions-per-cycle – IPC) ולא זמן ביצוע כולל כדי שניתן יהיה להשוות ביצועים של ריצות שונות של אותה תוכנית עם מספר חוטים (threads) שונה, מכיוון שמספר החוטים עשוי לגרום לתוכנית להשתמש בגודל בעיה שונה (ואנו יודעים שהתוכניות בחבילת Parsec אכן עושות זאת). מנגד, מדד זה איננו שימושי עבור תוכניות שמבצעות פעולות ספקולטיביות מכיוון שאז מספר גדול יותר של פעולות ליחידת זמן לאו דווקא מוביל מהר יותר לפתרון הבעיה.

בהינתן פרמטרי הארכיטקטורה והתוכנית, מודל אנליטי של הביצועים ניתן ע"י נוסחה (1):

$$\text{(1)}\quad \mathbf{Performance}(S_{\$},n)[IPC] = \frac{n}{CPI_{exe}+r_m\cdot\{P_{hit}(S_{\$},n)\cdot t_{\$}+[1-P_{hit}(S_{\$},n)]\cdot t_m\}}$$

מכיוון שביצועי זיכרון המטמון הפרטי (L1) לא מושפעים ממספר יחידות העיבוד, ההשפעה היחידה של זיכרון המטמון הפרטי על הביצועים במודל המתואר היא הקטנת החלק היחסי של הוראות מעבד הניגשות לנתונים בזיכרון מתוך סך כל ההוראות שהמעבד מבצע מנקודת המבט של זיכרון המטמון המשותף. לכן, השפעה זו יכולה להתבטא במודל ע"י התאמה של הפרמטר r_m.

הסימולטור

לצורך איפיון התנהגות של תוכניות אמיתיות תחת המודל המתואר פיתחנו סימולטור המממש את המודל, עם תוספת משמעותית: התוכנית מבוצעת בהתאם למודל התיזמונים כך שההשפעה של התיזמון על החישוב נשמרת. בפרט הסימולציה משמרת סנכרון והמתנה בין חוטים. המודל האנליטי לא מביא בחשבון תופעות אלו. כמו כן, פרמטרי המודל הם ממוצעים, כלומר המודל לא מביא בחשבון שינויים בהתנהגות התוכנית בזמן ובמרחב.

סימולציית התיזמון ממומשת בטכניקה של trace-driven simulation, כלומר מקבלת את זרם ההוראות של כל החוטים ואז מבצעת סימולציה של מודל התיזמון. סימולציה בטכניקה זו בד"כ לא משמרת השפעה של התיזמונים על החישוב כי החישוב כבר התבצע בזמן איסוף זרמי ההוראות שקודם לשלב סימולציית התיזמון. הסימולטור שלנו משמר את ההשפעה של מודל התיזמונים על החישוב ע"י מנגנון משוב בין איסוף זרמי ההוראות לבין סימולציית התיזמונים. משוב זה מושג ע"י כך שהסימולציה מבוצעת בלולאה בת שני שלבים:

1. סימולציית תיזמונים של ההוראות שנאספו עד כה.
2. איסוף זרמי ההוראות כל עוד ההוראה הבאה אינה כזאת המהווה תקשורת בין-חוטית (כמו למשל Compare-And-Swap – CAS) ולכן החישוב אינו מושפע מהתיזמון.

כאשר אחד הזרמים בוצע עד תומו ע"י סימולציית התיזמונים (שלב 1), משמע שלפי מודל התיזמון החוט המתאים הגיע ראשון מבין כל החוטים להוראה המהווה תקשורת בין-חוטית. בשלב זה סימולציית התיזמונים נעצרת (בהגדרה זרמי ההוראות של החוטים האחרים לא סומלצו עד תומם) ועוברים לאסוף זרם הוראות חדש מהחוט המתאים (שלב 2). זה נעשה ע"י כך שנותנים לחוט זה לרוץ וע"י כך לבצע את ההוראות הבאות (בפרט הראשונה מהווה תקשורת בין-חוטית) עד להוראה הבאה המהווה תקשורת בין-חוטית. בשלב זה איסוף זרם ההוראות נעצר ועוברים חזרה לשלב סימולציית התיזמונים וחוזר חלילה. בדרך זו כל ההוראות המהוות תקשורת בין-חוטית מבוצעות בסדר המוכתב ע"י מודל התיזמונים ולכן ההשפעה של מודל התיזמונים על החישוב (הביצוע הפונקציונלי) נשמרת.

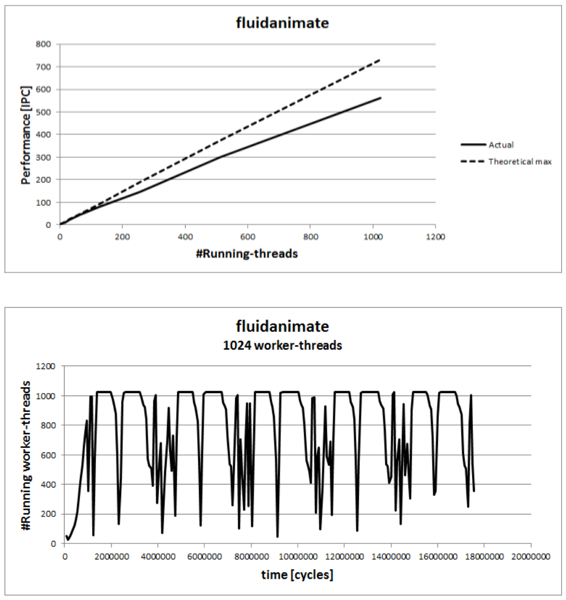
מדרגיות מקביליות (Parallelism scalability)

אנו מעוניינים לאפיין את היכולת של תוכניות נתונות לנצל ארכיטקטורות בעלות דרגות מקביליות הולכות וגדלות. איפיון זה מתבטא במידת העלייה בביצועים יחסית למידת העלייה בדרגת המקביליות של הארכיטקטורה. אנו מכנים תכונה זו *מדרגיות מקביליות של התוכנית*.

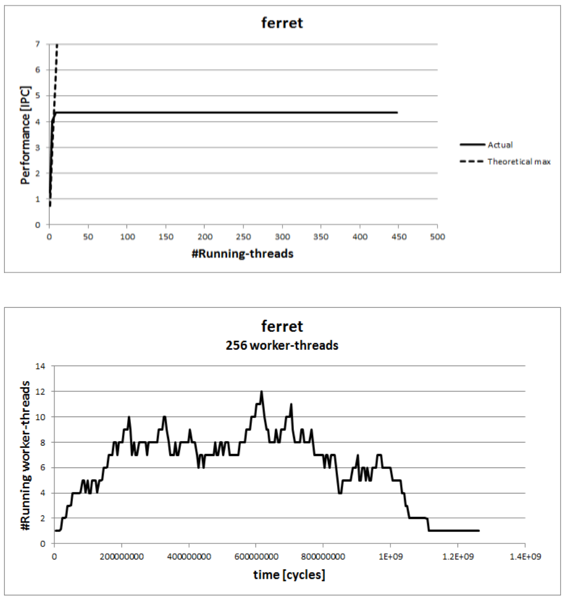
אנו משתמשים בסימולטור שלנו לאפיין את מדרגיות המקביליות המובנית של תוכניות בדיקה מקביליות (benchmarks) הנמצאות בשימוש נפוץ: Parsec benchmark suite. במדרגיות המקביליות המובנית הכוונה למדרגיות המקביליות של התוכנית במנותק מהארכיטקטורה המסוימת שעליה היא מתבצעת, העלולה בעצמה לפגוע במדרגיות המקביליות. לצורך כך אנו משתמשים במודל ארכיטקטוני ללא מגבלות מיקבול: מספר יחידות עיבוד לפחות כמספר החוטים בתוכנית ומערכת זיכרון אידאלית -- השהיה מינימלית של מחזור שעון אחד וללא מגבלות רוחב פס.

איור 2 ואיור 3 הם דוגמאות לתוכניות עם מדרגיות מקביליות טובה וגרועה, בהתאמה. עקומות הביצועים המקסימליים התיאורטיים (Theoretical max) מראות את הביצועים שהיו מושגים עם כל החוטים היו פעילים למשך כל זמן הריצה, כלומר חוט אף פעם לא נאלץ לחכות. עבור ferret רוב עקומת הביצועים המקסימליים התיאורטיים נמצאת מחוץ לטווח המוצג של הציר האנכי. אחרת עקומת הביצועים בפועל הייתה מתגמדת.

ההסבר למדרגיות המקביליות ניתן מתוך הגרפים של כמות החוטים הפעילים לאורך ריצת התוכנית. עבור fluidanimate ניתן לראות כי כל החוטים (1024) פעילים רוב הזמן. לעומת זאת עבור ferret ניתן לראות שלמרות שיש 256 חוטים, מספר החוטים הפעילים בו זמנית אף פעם לא עולה על 12.



איור 2: מדרגיות מקביליות טובה



איור 3: מדרגיות מקביליות גרועה

חקירת מודל אנליטי לביצועי זיכרון מטמון משותף

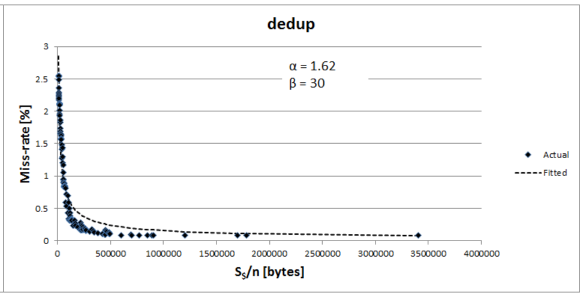
ב-[4] מוצע מודל אנליטי לביצועי זיכרון מטמון משותף (miss-rate) המתואר בנוסחה (2):

$$(2)\quad \mathbf{P_{miss}(S_\$, n)} = \left(\frac{\boldsymbol{S_\$}}{\boldsymbol{n\cdot\beta}} + \mathbf{1}\right)^{-(\boldsymbol{\alpha}-\mathbf{1})}$$

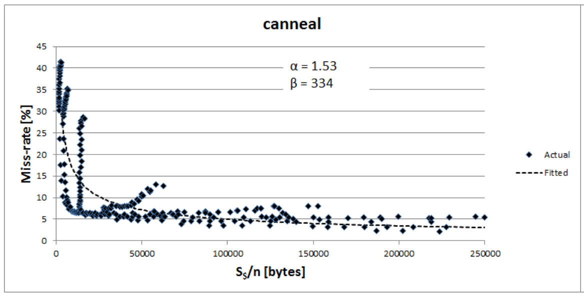
מודל זה מכיל פרמטרים α ו-β התלויים בתוכנית.

אנו משתמשים בסימולטור שלנו למדידת הביצועים בפועל (P*miss*) של תוכניות הבדיקה השונות שב-Parsec benchmark suite עם זיכרון מטמון משותף בגדלים שונים ($S_\$$) וכמויות חוטים שונות (n). באמצעות נתונים אלו ושימוש בשיטת Levenberg-Marquardt להתאמת עקומות (curve-fitting) אנו מוצאים את ערכי α ו-β המתאימים לכל תוכנית.

איור 4 ואיור 5 הם דוגמאות לתוכניות בעלות ביצועי זיכרון מטמון עם התאמה טובה וגרועה למודל האנליטי, בהתאמה.



איור 4: התאמה טובה למודל האנליטי



איור 5: התאמה גרועה למודל האנליטי

לרוב התוכניות שב-Parsec benchmark suite יש התאמה טובה יחסית לנוסחה (2) כך שנוסחה זו מהווה מודל מסדר ראשון טוב לביצועי זיכרון המטמון. אולם, כאשר משלבים אותה במודל האנליטי של ביצועי התוכנית שבנוסחה (1), עבור חלק מתוכניות הבדיקה מתקבלות סטיות גדולות בין הביצועים למרות התאמה טובה של נוסחה (2). דבר זה נובע מכך שנוסחה (1) רגישה יותר לשינויים בביצועי זיכרון המטמון ככל שה-miss-rate קטן יותר.

מדרגיות המקביליות של רוב תוכניות הבדיקה אינה משתנה עם זיכרון מטמון אולם יש כמה שמדרגיות המקביליות גדלה וכמה שבהם היא קטנה. ירידה במדרגיות המקביליות יכולה לנבוע למשל מכך שזיכרון המטמון הוא משאב משותף שהחוטים מתחרים עליו ותחרות זו יכולה לגרום לכך שתוספת הביצועים השולית עם הגידול במספר החוטים תיעשה זניחה (כלומר נקודת שיא הביצועים תתקבל במספר קטן יותר של חוטים). עליה במדרגיות המקביליות עם מערכת זיכרון לא מושלמת יחסית למדרגיות המקביליות עם מערכת זיכרון מושלמת יכולה לנבוע למשל מכך שהשיהוי הגדול יותר במקרה הראשון מגדיל את הזמן בין הגישות של חוט בודד למשאב משותף (כמו זיכרון המטמון) ובכך להפחית את התחרות עליו מה שמאפשר לו לשרת טוב יותר את החוטים שמספיקים להשתמש בו בזמן זה.