Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design

Stijn Eyerman   Lieven Eeckhout
Ghent University, Belgium
Email: {seyerman,leeckhou}@elis.UGent.be

ABSTRACT

This paper presents a fundamental law for parallel performance: it shows that parallel performance is not only limited by sequential code (as suggested by Amdahl's law) but is also fundamentally limited by synchronization through critical sections. Extending Amdahl's software model to include critical sections, we derive the surprising result that the impact of critical sections on parallel performance can be modeled as a completely sequential part and a completely parallel part. The sequential part is determined by the probability for entering a critical section and the contention probability (i.e., multiple threads wanting to enter the same critical section).

This fundamental result reveals at least three important insights for multicore design. (i) Asymmetric multicore processors deliver less performance benefits relative to symmetric processors than suggested by Amdahl's law, and in some cases even worse performance. (ii) Amdahl's law suggests many tiny cores for optimum performance in asymmetric processors, however, we find that fewer but larger small cores can yield substantially better performance. (iii) Executing critical sections on the big core can yield substantial speedups, however, performance is sensitive to the accuracy of the critical section contention predictor.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Modeling of computer architecture; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms

Performance

Keywords

Analytical performance modeling, Amdahl's law, synchronization, critical sections

1. INTRODUCTION

Amdahl's law, in spite of its simplicity, has important consequences for the multicore or chip-multiprocessor (CMP) era [10]. The software model in Amdahl's law is simple though: it assumes either completely sequential code or completely parallel code. This paper augments Amdahl's software model with the notion of synchronization through critical sections. Assuming that part of the parallel code needs to be executed within critical sections and assuming that critical sections are entered at random times, we derive a simple analytical (probabilistic) model for how long it takes to execute a critical section (including the time waiting for the critical section to be released). This simple model reveals a novel and surprising insight: it shows that the time spent in critical sections can be modeled as a completely sequential part plus a completely parallel part. The sequential part is proportional to the probability for a critical section and the contention probability (multiple threads competing for the same critical section).

Augmenting Amdahl's law with this probabilistic model for critical sections reveals that parallel performance is not only limited by sequential code (as suggested by Amdahl's law) but is also fundamentally limited by synchronization. Although it is well understood that fine-grain synchronization and low contention rates lead to better performance, this paper is the first, to the best of our knowledge, to provide a theoretical underpinning.

This fundamental law has important implications for multicore design. For one, asymmetric multicore processors (with one big core for sequential code and many small cores for parallel code) offer smaller performance benefits relative to symmetric multicore processors than suggested by Amdahl's law. In some cases, asymmetric processors may even yield worse performance than symmetric processors, the reason being that contending critical sections are executed on the small cores rather than on the relatively larger cores in a symmetric multicore processor. Second, Amdahl's law suggests that optimum performance is achieved with many tiny cores in an asymmetric processor, however, we find it is beneficial to make these cores larger, and thus have fewer cores for the same hardware budget. Larger small cores speed up the execution of critical sections, thereby improving overall parallel performance (although there are fewer cores). Third, accelerating critical sections (ACS) [22] by migrating critical sections to the big core can yield substantial speedups, provided that a highly accurate critical section contention predictor is available. Naive ACS which migrates all critical sections to the big core results in false serialization which deteriorates performance, especially for low contention probabilities. The gap with perfect ACS, which executes only contending critical sections on the big core, is huge. Future research in ACS contention prediction is likely to yield substantial perfor-

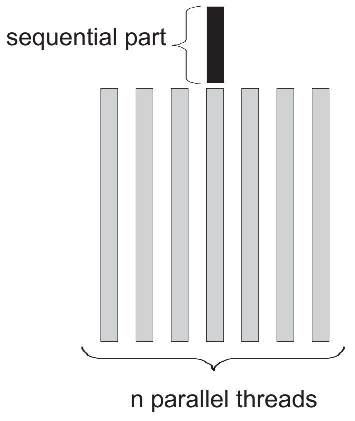

Figure 1: Amdahl's software model.

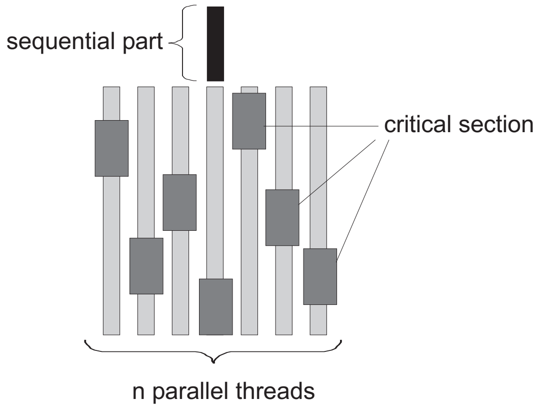

Figure 2: Amdahl's software model including critical sections.

mance benefits, and should focus on both eliminating false serialization and predicting contending critical sections.

2. AMDAHL'S LAW AND EXTENSIONS

We first briefly revisit Amdahl's law.

2.1 Amdahl's law

Define speedup as the original execution time divided by the enhanced execution time, and assume that a fraction f of a program's execution time is indefinitely parallelizable. Amdahl's law [1] then provides a formula for the speedup S that can be achieved by parallelizing the fraction f across n processors (see also Figure 1); the remaining fraction $(1-f)$ is executed sequentially:

$$S = \frac{1}{(1-f) + \frac{f}{n}}. \quad (1)$$

The significance of Amdahl's law for parallel computing is that it shows that the speedup is bound by a program's sequential part:

$$\lim_{n\to\infty} S = \frac{1}{1-f}. \quad (2)$$

2.2 Amdahl's law for asymmetric CMPs

Asymmetric multicore processors have one or more cores that are more powerful than the others [2, 4, 6, 10, 13]. Amdahl's law suggests that only one core should get more resources; the others should be as small as possible. The reason is that the parallel part gets a linear speedup from adding cores; the sequential part gets only sublinear speedup, i.e., core performance increases sublinearly with resources [5]. Assume now that the parallel part is executed on the n small cores at performance of one; further, assume that the sequential part is executed on the big core which yields a speedup of p relative to small core execution. Hence, asymmetric multicore speedup is computed as [10]:

$$S = \frac{1}{\frac{1-f}{p} + \frac{f}{n+p}}. \quad (3)$$

3. MODELING CRITICAL SECTIONS

Amdahl assumes a simple parallel software model: the program's execution time is either parallelizable or it is totally sequential. Amdahl's law does not take into account the impact of critical sections. We now extend Amdahl's law to model synchronization among parallel threads through critical sections.

Consider a program's sequential execution, and split it up in three fractions (see also Figure 2). We define f_{seq} as the totally sequential fraction of the program's execution that cannot be parallelized; this is the time required to spawn threads, distribute the data across the parallel threads, merge the data produced by the parallel worker threads, etc. Further, we define $f_{par,ncs}$ as the parallelizable fraction of the program's execution; this fraction is assumed to be indefinitely parallelizable, and does not need synchronization and is thus executed outside critical sections. Finally, we define $f_{par,cs}$ as the fraction of the program's execution time that can be parallelized but requires synchronization and is to be executed inside critical sections. If two or more threads compete for entering the same critical section at the same time, their execution will be serialized; if not, their execution can proceed in parallel. We define these fractions to sum up to one, i.e., $f_{seq} + f_{par,ncs} + f_{par,cs} = 1$, i.e., the fractions are all with respect to sequential execution.

3.1 Total execution time

We now compute what the execution time will be on a parallel machine. We therefore make a distinction between two cases, based on whether the total execution time can be approximated by the average per-thread execution time (i.e., all threads execute equally long, hence the total execution time can be approximated by the 'average' thread), or whether the total execution time is determined by the slowest thread (i.e., the time spent in critical sections is significant and the average thread is not representative for computing the total execution time).

3.1.1 Case #1: Execution time determined by average thread

The sequential part does not scale with parallelism ($T_{seq} \propto f_{seq}$); the parallel fraction executed outside critical sections on the other hand is indefinitely parallelizable ($T_{par,ncs} \propto f_{par,ncs}/n$). The difficulty now is to determine how long it will take to execute the parallel fraction due to critical sections ($T_{par,cs}$). For computing the average time spent in critical sections $T_{par,cs}$, we propose an analytical, probabilistic model.

We define the *probability for a critical section* during parallel execution P_{cs} as

$$P_{cs} = Pr[\textit{critical section} \mid \textit{parallel}] = \frac{f_{par,cs}}{f_{par,ncs} + f_{par,cs}} \quad (4)$$

The probability for i threads out of n threads to be executing in critical sections is binomially distributed, assuming that critical sections are entered at random times — this is an example of a

Bernoulli experiment — hence:

$$Pr[\textit{i of n threads in critical section}] = \binom{n}{i} P_{cs}^{i}\,(1-P_{cs})^{n-i} \tag{5}$$

We now define the *contention probability* P_{ctn}, or the probability for *two* critical sections to contend, i.e., both critical sections access the same shared memory state and hence they require serial execution, i.e., both critical sections are guarded by the same lock variable or both critical sections are executed within contending transactions.[1] Assuming that critical sections contend randomly, the probability for j threads out of these i threads to contend for the same critical section is also binomially distributed:

$$Pr[\textit{j of i threads contend for the same critical section}] = \binom{i}{j} P_{ctn}^{j}\,(1-P_{ctn})^{i-j} \tag{6}$$

We are now ready to compute the average time spent in critical sections. Consider a thread entering a critical section. The probability for i out of the remaining $n-1$ threads to also enter a critical section is determined by Equation 5; the probability for j out of these i threads to contend with the given critical section is determined by Equation 6. If j other critical sections contend with the current critical section, then in total $j+1$ critical sections will serialize, and hence it takes $(j+1)f_{par,cs}/n$ time units to execute this critical section: $j \cdot f_{par,cs}/n$ time units for executing the other critical sections (while the given critical section waits for the critical section to be released), plus $f_{par,cs}/n$ time units to execute the given critical section (once the thread acquires the critical section). Putting it all together, the average time spent in a critical section is computed as:

$$\begin{aligned} T_{par,cs} \propto \sum_{i=0}^{n-1} \binom{n-1}{i} P_{cs}^{i}\,(1-P_{cs})^{n-1-i} \cdot \\ \sum_{j=0}^{i} \binom{i}{j} P_{ctn}^{j}\,(1-P_{ctn})^{i-j} \cdot \frac{j+1}{n} \cdot f_{par,cs} \end{aligned} \tag{7}$$

Exploiting the property $\sum_{i=0}^{n}\binom{n}{i}P^{i}(1-P)^{n-i} = nP$ (average of a binomial distribution), we can simplify this expression to the following equation:

$$T_{par,cs} \propto f_{par,cs} \cdot \left(P_{cs}P_{ctn} + (1-P_{cs}P_{ctn}) \cdot \frac{1}{n}\right) \tag{8}$$

This expression reveals a novel, interesting and surprising insight: it shows that the average time spent in critical sections can be modeled as a completely sequential part plus a completely parallel part (see also Figure 3 for a graphical illustration). The sequential part is proportional to the probability for a critical section multiplied by the contention probability. The parallel part can be significant if the critical sections are small and show low contention probabilities.

Assuming that the total execution time of a parallel program can be approximated by the execution time of the 'average' thread, then the total execution time is proportional to

$$T \propto f_{seq} + f_{par,cs}P_{cs}P_{ctn} + \frac{f_{par,cs}(1-P_{cs}P_{ctn}) + f_{par,ncs}}{n} \tag{9}$$

[1]For a parallel program with only one lock variable for all critical sections, $P_{ctn} = 1$ and hence all critical sections will serialize; P_{ctn} is smaller than one for fine-grained locking with multiple lock variables and/or through rarely contending transactions.

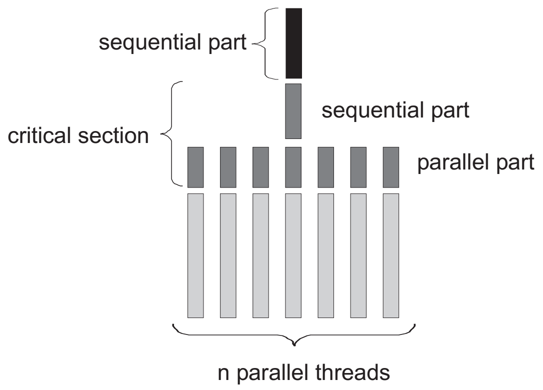

Figure 3: Critical sections can be modeled as a sequential part plus a parallel part.

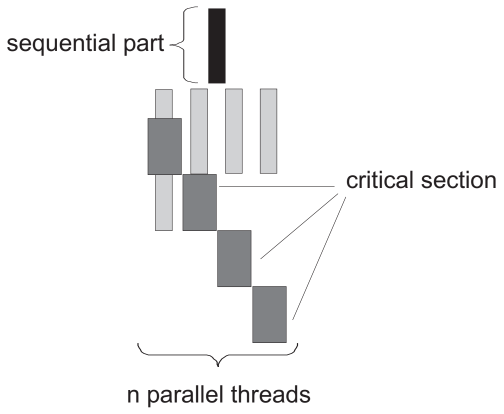

Figure 4: Example illustrating the case in which the total execution time is determined by the slowest thread.

3.1.2 *Case #2: Execution time determined by slowest thread*

The assumption that a parallel program's execution time equals the average per-thread execution time does not always hold true, especially in case the time spent in critical sections is high and/or contention probability is high — this is a case in which the total execution time is determined by the slowest thread, see Figure 4 for an example. We now derive a formula for the total execution time as the execution time of the slowest thread — we assume that the execution of the slowest thread is determined by the serialized execution of all the contending critical sections. The average execution time of the slowest thread is determined by three components: (i) the sequential part ($T_{seq} \propto f_{seq}$), (ii) the total time spent in contending critical sections which involves sequential execution ($T_{seq,cs} \propto f_{par,cs}P_{ctn}$), and (iii) the time spent executing parallel code including indefinitely parallel code and non-contending critical sections ($T_{par} \propto (f_{par,cs}(1-P_{ctn}) + f_{par,cs})/2n$; given that we assume that critical section occur randomly, we need to add the factor 2 in the denominator, i.e., on average half the parallel code will be executed before the contending critical sections and half the parallel code will be executed after the contending critical sections). The total execution time then equals the average execution time of the slowest thread:

$$T \propto f_{seq} + f_{par,cs}P_{ctn} + \frac{f_{par,cs}(1-P_{ctn}) + f_{par,ncs}}{2n} \tag{10}$$

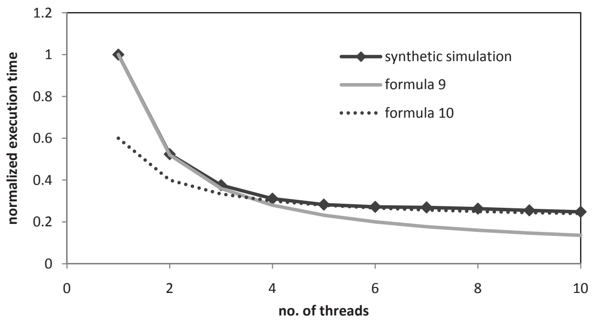

Figure 5: Validation of the analytical probabilistic model against a synthetic simulator.

3.1.3 *Putting it together*

We now have two expressions for the total execution time, each expression with its own set of assumptions. The former (case #1) assumes that the total execution time of the parallel program can be approximated by the average per-thread execution time; the latter (case #2) makes the observation that this assumption no longer holds true once the serialized execution of contending critical sections exceeds the average per-thread execution time. Hence, we conjecture the overall execution time can be computed as the maximum of these two expressions:

$$\begin{aligned} T \propto & f_{seq} + \\ & \max\Big(f_{par,cs}P_{cs}P_{ctn} + \frac{f_{par,cs}(1-P_{cs}P_{ctn}) + f_{par,ncs}}{n}; \\ & f_{par,cs}P_{ctn} + \frac{f_{par,cs}(1-P_{ctn}) + f_{par,ncs}}{2n}\Big) \end{aligned} \tag{11}$$

Because this formula is fairly complicated and builds on assumptions and probabilistic theory, we validate the formula against a synthetic simulator that generates and simulates a synthetic parallel workload. The validation was done across a wide range of possible values for $f_{par,cs}$, $f_{par,ncs}$, P_{ctn} and n; the average absolute error across all of these experiments equals 3%. Figure 5 shows the execution time as a function of the number of threads n for $f_{par,cs} = 0.5$, $f_{par,ncs} = 0.5$ and $P_{ctn} = 0.5$. For a small number of threads, then formula 9 (case #1) is more accurate, and for a large number of threads then formula 10 (case #2) is more accurate and converges towards 0.25 which equals the $f_{par,cs} \cdot P_{ctn} = 0.5 \cdot 0.5 = 0.25$. This graph illustrates that the max operator is a reasonable approximation.

3.2 Extending Amdahl's law

Integrating the above in Amdahl's law yields the following result for $n \to \infty$:

$$\begin{aligned} \lim_{n\to\infty} S &= \frac{1}{f_{seq} + \max(f_{par,cs}P_{cs}P_{ctn}; f_{par,cs}P_{ctn})} \\ &= \frac{1}{f_{seq} + f_{par,cs}P_{ctn}} \end{aligned} \tag{12}$$

This is a novel fundamental law for parallel performance: it shows that parallel speedup is not only limited by the sequential part (as suggested by Amdahl's law) but is also fundamentally limited by synchronization. In other words, the larger the time spent in critical sections, the lower the maximum speedup. Similarly, the larger the contention probability, the lower the maximum speedup. Although it is intuitively well understood that fine-grained critical sections and low contention rates lead to better performance, this paper is the first to provide a theoretical underpinning for this insight. It shows that parallel performance is fundamentally limited by synchronization, and more specifically, critical section size and their contention probability.

4. IMPLICATIONS FOR CMP DESIGN

The above fundamental finding has important implications for multicore design, especially in the context of asymmetric multicore processors.

4.1 Asymmetric multicores

One compelling argument for asymmetric multicore processors is that the sequential fraction of a program's execution can be executed on the big core whereas the parallel work can be done on the small cores [2, 3, 4, 10, 15]. Given Amdahl's law, this may yield substantially higher speedups compared to symmetric multicore processors.

Critical sections however limit the potential benefit from asymmetric multicore processors, because the parallel part of the program's execution time is not indefinitely parallelizable. Instead there is a sequential part due to critical sections that will be executed on the small cores. Assuming n small cores with performance of one, along with a big core that yields a speedup of p compared to the small cores, the execution time on an asymmetric processor is:

$$\begin{aligned} T \propto & \frac{f_{seq}}{p} + \\ & \max\Big(f_{par,cs}P_{cs}P_{ctn} + \frac{f_{par,cs}(1-P_{cs}P_{ctn}) + f_{par,ncs}}{n+p}; \\ & f_{par,cs}P_{ctn} + \frac{f_{par,cs}(1-P_{ctn}) + f_{par,ncs}}{2(n+p)}\Big) \end{aligned} \tag{13}$$

The key insight from this equation is that the sequential part due to critical sections has an even bigger (relative) impact on performance for an asymmetric multicore processor than is the case for a symmetric multicore processor. This is because the sequential fraction of the execution time is sped up on the asymmetric processor (by a factor p) relative to the symmetric processor which is not the case for the sequential fraction due to critical sections (compare Equation 13 against Equation 11). The impact of sequential execution of critical sections increases further with increasing probability for a critical section and contention probability.

This, in its turn, has an important implication on how to design an asymmetric multicore processor. For asymmetric multicore processors, the original Amdahl's law suggests that the maximum speedup is achieved with the small cores being as small as possible. Tiny cores imply that more cores can be put on the die, which yields a linear performance speedup for parallel code; larger cores (and thus fewer cores) do not yield as much performance benefit because of sublinear core performance improvements with resources [5]. Now, the new Amdahl's law for asymmetric processors (Equation 13) which includes the effect of critical sections, suggests that the small cores in asymmetric processors should be sufficiently large (and not be as small as possible as suggested by Amdahl's law) because they will be executing the sequential part of the critical sections. In other words, it is important that critical sections be executed fast so that other threads can enter their critical sections as soon as possible.

To verify this intuition, we now consider a simple design problem in which we have a total number of 256 base core equivalents

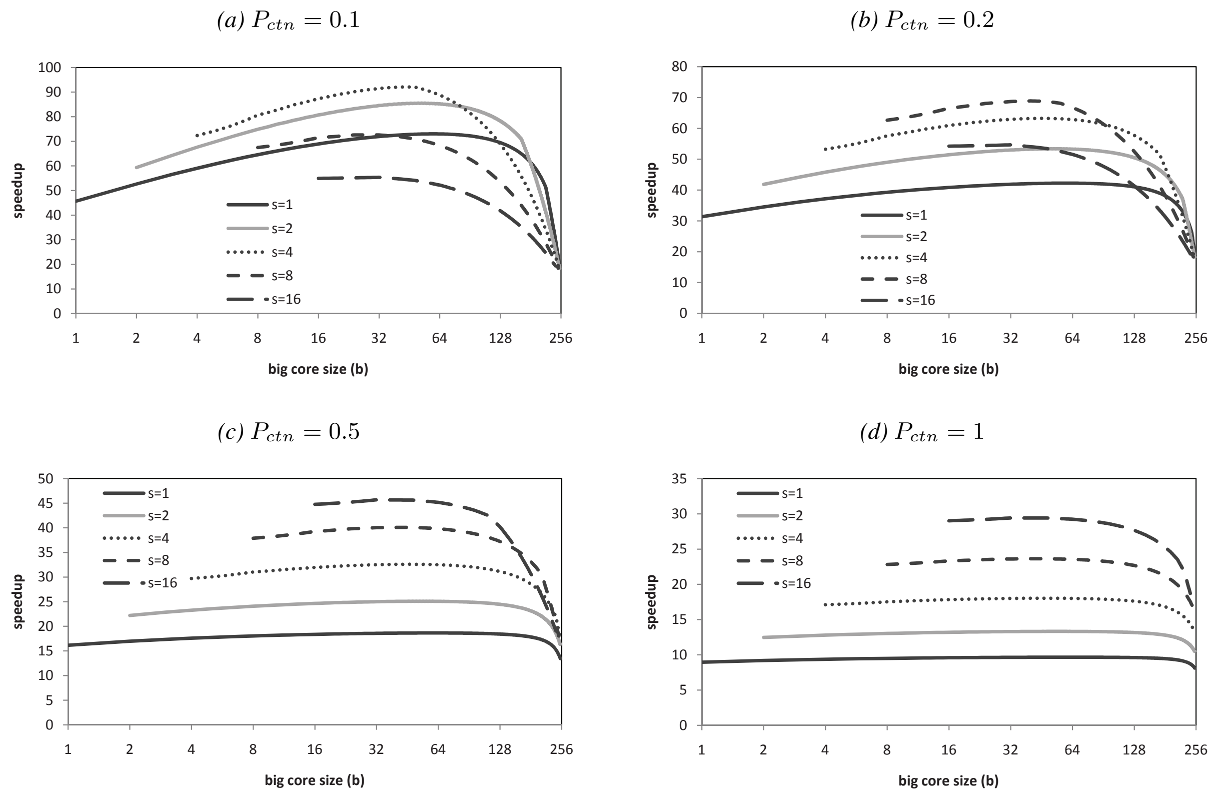

Figure 6: Speedup for an asymmetric processor as a function of the big core size (b) and small core size (s) for different contention rates, assuming a total number of 256 BCEs.

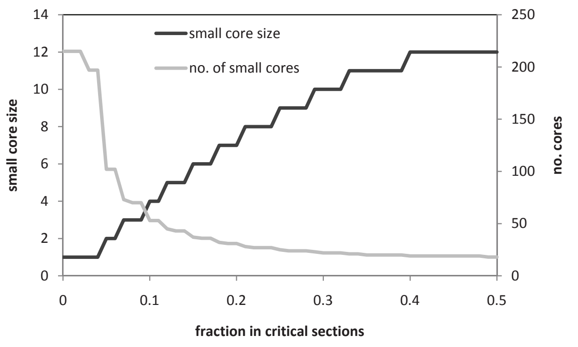

Figure 7: Optimum number of small cores and their size as a function of the fraction of time spent in critical sections, assuming $P_{ctn} = 0.1$.

(BCEs), following the approach taken by Hill and Marty [10]. We now divide these BCEs among the big core (b BCEs) and the small cores (s BCEs per core). Figure 6 shows the attainable speedup as a function of the size of the big core (b) and the size of the small cores (s); we assume a square-root relationship between resources (number of BCEs) and performance [5] (also known as Pollack's law). For low contention rates, small cores yield the best speedup (see Figure 6(a)). However, for higher contention rates, larger cores yield the best performance. For example, for a contention rate of 20% (Figure 6(b)), optimum performance is achieved for a large core of size 40 BCEs and 27 small cores of size 8 BCEs. Figure 7 further confirms this finding and shows the optimum number of small cores and their size as a function of $f_{par,cs}$, or the fraction of the program's execution in critical sections. The larger the fraction spent in critical sections, the larger the small cores should be, and (hence) the fewer small cores there need to be for optimum performance.

4.2 Accelerating critical sections

A solution to mitigate the large impact of critical sections on parallel performance, is to execute the critical sections on the big core in an asymmetric multicore processor. The basic idea is that if critical sections are released earlier, other threads can enter the critical section faster. This technique is called Accelerating Critical Sections (ACS) [22].

4.2.1 *Naive ACS*

Naive ACS would execute all critical sections serially on the large core, which yields the following execution time:

$$T \propto \frac{f_{seq} + f_{par,cs}}{p} + \frac{f_{par,ncs}}{n+p}. \tag{14}$$

Compared to Equation 13, we observe that the sequential part due to synchronization is sped up by a factor p, however, the parallel part gets slowed down by a factor $(n+p)/p$. Naive ACS may lead to serialized execution of non-contending critical sections (false serialization) which potentially has a detrimental effect on performance.

4.2.2 *Perfect ACS*

Perfect ACS, which would yield optimum ACS performance, would execute only the sequential part due to synchronization (i.e., the contending critical sections) on the big core; the parallel part (including the non-contending critical sections) would still be executed on the small cores. The execution time under perfect ACS equals:

$$\begin{aligned} T \propto & \frac{f_{seq}}{p} + \\ & \max\Big(\frac{f_{par,cs}P_{cs}P_{ctn}}{p} + \frac{f_{par,cs}(1-P_{cs}P_{ctn}) + f_{par,ncs}}{n+p}; \\ & \frac{f_{par,cs}P_{ctn}}{p} + \frac{f_{par,cs}(1-P_{ctn}) + f_{par,ncs}}{2(n+p)}\Big) \end{aligned} \tag{15}$$

Perfect ACS is hard (if not impossible) to achieve because it requires a predictor that perfectly predicts whether critical sections will contend (and thus need to be serialized and executed on the big core).

4.2.3 *Selective ACS*

Suleman et al. [22] also recognized that in order to mitigate false serialization, one needs to predict whether critical sections will contend or not — this is called Selective ACS. Predicting whether critical sections will contend or not is non-trivial, and mispredictions have significant impact on performance. Predicting that a critical section will contend and needs to be executed on the big core, which eventually does not contend — a false positive — results in false serialization. Predicting that a critical section will not contend and execute that critical section on a small core, which eventually results in a contention — a false negative — unnecessarily prolongs the critical section's execution. Assuming a contention predictor with sensitivity α (ability to predict true positives) and specificity β (ability to predict true negatives), the execution time under selective ACS is given by (define $P_{cf} = P_{cs}P_{ctn}$):

$$\begin{aligned} T \propto & \frac{f_{seq}}{p} + \\ & \max\Big(\frac{f_{par,cs}(\alpha P_{cf} + (1-\beta)(1-P_{cf}))}{p} + \\ & (1-\alpha)f_{par,cs}P_{cf} + \frac{f_{par,cs}\beta(1-P_{cf}) + f_{par,ncs}}{n+p}; \\ & \frac{f_{par,cs}(\alpha P_{ctn} + (1-\beta)(1-P_{ctn}))}{p} + \\ & (1-\alpha)f_{par,cs}P_{ctn} + \frac{f_{par,cs}\beta(1-P_{ctn}) + f_{par,ncs}}{2(n+p)}\Big) \end{aligned} \tag{16}$$

Note that for $\alpha = 1$ and $\beta = 0$, selective ACS equals naive ACS; for $\alpha = 0$ and $\beta = 1$, selective ACS equals a conventional asymmetric multicore processor (see Equation 13); perfect ACS means $\alpha = 1$ and $\beta = 1$. We will evaluate ACS performance and its dependence on specificity and sensitivity as part of the next section.

4.3 High-level CMP design space exploration

Having obtained the above formulas, we can now perform a high-level design space exploration for multicore processors in which we compare symmetric multicore processors and asymmetric multicore processors against naive, perfect ACS and selective ACS, while varying the number of cores, their size and the fraction of the program's execution time spent in critical sections. We will vary the fraction of the time spent in critical sections and their contention rates, and reason about which processor architecture yields the best performance. Similar as before, we assume 256 BCEs and a square-root relationship between resources (number of BCEs) and core-level performance [5]. We consider the following processor architectures:

- **Symmetric multicore.** The symmetric multicore processor is optimized for maximum speedup, i.e., we optimize the number of cores (and their size) for optimum speedup.
- **Asymmetric multicore with tiny cores.** The asymmetric multicore processor has tiny small cores (single-BCE with performance of one). We optimize the big core for optimum speedup. This corresponds to the asymmetric processor assumed by Hill and Marty [10], and is the optimum asymmetric processor according to the original Amdahl's law.
- **Asymmetric multicore with optimized cores.** Both the big core and the small cores are optimized for optimum performance. The small cores are potentially larger than a single BCE in order to execute serialized critical sections faster.
- **Naive ACS.** The big core is optimized for optimum performance; the optimum small core size is one BCE. All critical sections are executed on the big core.
- **Perfect ACS.** Same as for naive ACS, however, only the contending critical sections are serialized on the big core; the non-contending critical sections are executed in parallel on the small cores.

Figure 8 shows the speedup for each of these processor architectures for different contention rates (different graphs show different contention rates) and different fractions of the time spent in critical sections ($f_{par,cs}$ on the horizontal axes). One percent of the code is completely sequential ($f_{seq} = 0.01$). We derive some interesting insights.

Asymmetric versus symmetric multicore processors. The benefit of asymmetric processing over symmetric processing is limited by critical sections (see the symmetric and asymmetric ($s = 1$) curves in Figure 8). In the absence of critical sections, asymmetric processors offer a substantial performance benefits over symmetric processors, see the leftmost points on the graphs ($f_{par,cs} = 0$) — these points correspond to the data presented by Hill and Marty [10]. However, as the time spent in critical sections and their contention rate increases, the relative benefit from asymmetric multicore processing decreases. For large critical sections and high contention rates, symmetric multicore processors even yield worse performance compared to asymmetric processors with tiny small cores. The reason is that the contending critical sections serialize on the small single-BCE cores in the asymmetric processor; on a symmetric processor, the critical sections are executed on relatively larger cores.

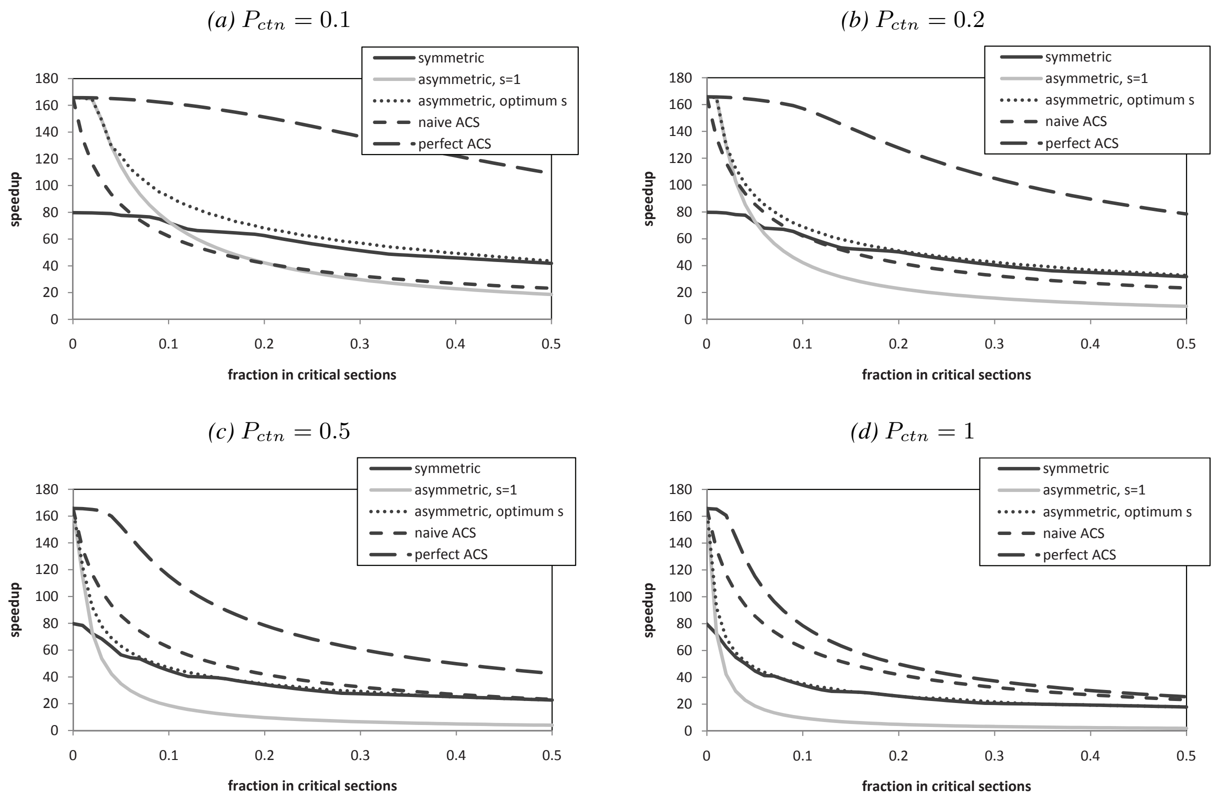

Figure 8: High-level design space exploration across symmetric, asymmetric and ACS multicore processors while varying the fraction of the time spent in critical sections and their contention rates. Fraction spent in sequential code equals 1%.

Because critical sections serialize on the small cores, it is beneficial to increase the small core size to be larger than a single BCE. By optimizing the big core and small core sizes, one can optimize asymmetric multicore processor performance to achieve the best of both worlds and achieve at least as good performance as symmetric processors and asymmetric processors with single-BCE small cores (see the asymmetric curve with optimum s in Figure 8).

Naive ACS. Naive ACS serializes all critical sections on the big core, even though these critical sections may not contend with each other (false serialization). Naive ACS performance quickly deteriorates for even small increases in the time spent in critical sections, see left part of the graphs ($0 < f_{par,cs} < 0.1$). Naive ACS performs worse than an asymmetric processor with optimum small core size for low contention rates (see $P_{ctn} = 0.1$), however, for larger contention rates, naive ACS performs better than asymmetric (and symmetric) processors. This is because naive ACS executes all critical sections on the big core, whereas the asymmetric and symmetric processors execute these critical sections on relatively small cores.

Huge gap between naive and perfect ACS. There is a huge gap between naive and perfect ACS for low contention probabilities (see $P_{ctn} = 0.1$ and $P_{ctn} = 0.2$). Perfect ACS predicts whether or not critical sections will contend; if they do contend, the critical sections are executed on the big core; if not, they are executed in parallel on the small cores. This suggests that significant performance benefits are to be gained from improving ACS contention predictors — a good avenue for future work. However, for higher contention probabilities, this gap becomes smaller (see $P_{ctn} = 0.5$ and $P_{ctn} = 1$), and naive ACS is likely to achieve most of the benefits of perfect ACS.

Selective ACS. Figure 9 shows speedup as a function of the ACS contention predictor's specificity and sensitivity, and fraction in critical sections $f_{par,cs}$ on the horizontal axis. The important insight here is that for selective ACS to yield speedup compared to naive ACS, the contention predictor's sensitivity is more important than its specificity. (ACS with $\alpha = 0.98$ and $\beta = 0.50$ yields better performance than naive ACS across all critical section sizes.) Hence, the ability to predict contending critical sections (that will be executed on the big core) is more important than eliminating false serialization (i.e., executing non-contending critical sections on the small cores). Optimizing and balancing both sensitivity and specificity yields even better performance (see the curve with $\alpha = 0.85$ and $\beta = 0.85$ in Figure 9), especially for smaller fractions of time spent in critical sections.

5. LIMITATIONS

Arguably, the synchronization model presented in this paper has its limitations. For one, it is limited to critical sections only and does not tackle other forms of synchronization. Further, it assumes that the parallel workload is homogeneous, i.e., all threads execute the same code and exhibit the same probability for a critical section and the same contention probability. We also assume that the critical sections are entered at random times. The original Amdahl's law and its extension for multicore hardware also has its limitations, as discussed by Hill and Marty [10]. Some researchers even question the validity of Amdahl's law, see for example [7, 19]. However, and more importantly though, the goal of the model is not to present accurate quantitative performance numbers and predictions, but to provide insight and intuition, and stimulate discussion and future work.

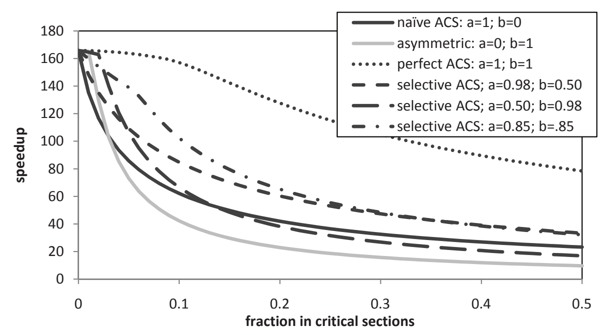

Figure 9: Speedup as a function of the ACS contention predictor's specificity and sensitivity. Fraction spent in sequential code equals 1%, and the contention probability equals 0.2.

6. RELATED WORK

6.1 Asymmetric processors

Asymmetric processors have been proposed to improve performance by many researchers, see for example [10, 15, 18]. As suggested by Amdahl's law, the big core executes the sequential part whereas the parallel part is executed on the small cores. None of these prior works incorporated the notion of critical sections though. However, when incorporated in Amdahl's law, critical sections significantly change the view on asymmetric processors in a number of ways — as described extensively in this paper.

6.2 Other architecture paradigms

The insights from this paper could also inspire other architecture paradigms. For example, multicore processors with each core supporting simultaneous multithreading (SMT) [24] could benefit from a critical section contention predictor in a similar way as we discussed for ACS on asymmetric multicores. The thread entering a predicted contending critical section could be given more or even all fetch slots by the SMT fetch policy (i.e., throttle the other threads), so that the critical section can execute faster and is not slowed down by other co-executing threads. Core fusion [11] can benefit in a similar way: the critical section contention predictor can predict when to fuse multiple small cores to form a more powerful core at runtime. Also, a conflict contention predictor would enable applying core-level DVFS [12] on a multicore processor to contending critical sections which would optimize performance (execute the critical sections faster) while limiting power consumption (by not boosting non-contending critical sections).

6.3 Feedback-driven threading

Suleman et al. [23] propose feedback-driven threading, a framework for dynamically controlling the number of threads during runtime. They propose a simple analytical model to derive the optimum number of threads. The model proposed for applications that are bounded by synchronization is similar to the model discussed in Section 3.1.2 which assumes that the total execution time is determined by the slowest thread and the serialized execution of the contending critical sections.

6.4 Reducing impact of critical sections

Reducing the impact of critical sections on parallel performance is a very active field of research. It is well known that fine-grain synchronization (i.e., small critical sections) generally lead to better performance than coarse-grain synchronization (i.e., large critical sections) — at the cost of significant programming effort. The critical section model presented in this paper provides a theoretical underpinning for this well-known insight: critical section size relates to the probability for a critical section P_{cs} and has a substantial impact on parallel performance.

Several recent enhancements reduce contention (P_{ctn}) compared to lock-based critical sections. Techniques such as transactional memory (TM) [8, 17, 9], speculative lock elision [20], transactional lock removal [21] and speculative synchronization [14] execute critical sections speculatively without acquiring the lock, and if they do not access the same shared data, they are committed successfully.

6.5 Workload characterization

Stanford's TCC group proposed and characterized the STAMP TM benchmark suite [16]. They found that several of these benchmarks spend a substantial amount of time in critical sections (transactions) — this relates to P_{cs} in our model. Only 2 of the 8 benchmarks spend less than 20% in transactions, one benchmarks spends around 40% of its time in transactions, and the remaining 5 benchmarks spend over 80% and up to 100% in transactions. The paper does not report contention probabilities, but instead reports the number of retries per transaction: depending on the TM system, some benchmarks exhibit anything between zero retries (no contention) up to 10 retries per transaction. We thus believe that the values assumed in this paper for P_{cs} and P_{ctn} are reasonably realistic.

7. CONCLUSION

This paper augments Amdahl's law with the notion of critical sections. A simple analytical (probabilistic) model reveals that the impact of critical sections can be split up in a completely sequential and a completely parallel part. This leads to a new fundamental law: parallel performance is not only limited by the sequential part (as suggested by Amdahl's law) but is also fundamentally limited by critical sections. This result provides a theoretical underpinning for research efforts towards fine-grain synchronization and low contention rates.

This fundamental result has important implications for asymmetric multicore processor design. (i) The performance benefit of asymmetric multicore processors may not be as high as suggested by Amdahl's law, and may even be worse than symmetric multicore processors for workloads with many and large critical sections and high contention probabilities. (ii) The small cores should not be tiny as suggested by Amdahl's law, but should instead be larger to execute critical sections faster. (iii) Accelerating critical sections through execution on the big core may yield substantial speedups, however, performance is very sensitive to the critical section contention predictor.

8. ACKNOWLEDGEMENTS

We thank Aater Suleman, Mark Hill and the anonymous reviewers for their constructive and insightful feedback. Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0232.06, G.0255.08, and G.0179.10, and the UGent-BOF projects 01J14407 and 01Z04109.

9. REFERENCES

[1] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings*

of the American Federation of Information Processing Societies Conference (AFIPS), pages 483–485, 1967.

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

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 506–517, June 2005.

[5] S. Borkar. Thousand core chips — a technology perspective. In *Proceedings of the Design Automation Conference (DAC)*, pages 746–749, June 2007.

[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March/April 2006.

[7] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. D. an B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 102–113, June 2004.

[9] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, June 1993.

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

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

[12] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 123–134, Feb. 2008.

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

[14] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, Oct. 2002.

[15] D. Menasce and V. Almeida. Cost-performance analysis of heterogeneity in supercomputer architectures. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 169–177, Nov. 1990.

[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 35–46, Sept. 2008.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 254–265, Feb. 2006.

[18] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and A. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, Jan. 2006.

[19] J. M. Paul and B. H. Meyer. Amdahl's law revisited for single chip systems. *International Jounal of Parallel Programming*, 35(2):101–123, Apr. 2007.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 294–305, Dec. 2001.

[21] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, Oct. 2002.

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

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

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