International Journal of Parallel Programming, Vol. 35, No. 2, April 2007 (© 2007)
DOI: 10.1007/s10766-006-0028-8

Amdahl's Law Revisited for Single Chip Systems

JoAnn M. Paul[1,3] and Brett H. Meyer[2]

Received August 4, 2006; revised October 18, 2006; accepted November 17, 2006

Amdahl's Law is based upon two assumptions – that of boundlessness and homogeneity – and so it can fail when applied to single chip heterogeneous multiprocessor designs, and even microarchitecture. We show that a performance increase in one part of the system can negatively impact the overall performance of the system, in direct contradiction to the way Amdahl's Law is instructed. Fundamental assumptions that are consistent with Amdahl's Law are a heavily ingrained part of our computing design culture, for research as well as design. This paper points in a new direction. We motivate that emphasis should be made on holistic, system level views instead of divide and conquer approaches. This, in turn, has relevance to the potential impacts of custom processors, system-level scheduling strategies and the way systems are partitioned. We realize that Amdahl's Law is one of the few, fundamental laws of computing. However, its very power is in its simplicity, and if that simplicity is carried over to future systems, we believe that it will impede the potential of future computing systems.

KEY WORDS: Amdahl's Law; single chip heterogeneous multiprocessing; performance; design; partitioning.

1. INTRODUCTION

Future single chip designs for portable and handheld computers are expected to be multiprocessors, built around sets of heterogeneous

[1]ECE Department (Advanced Research Institute), Virginia Tech, 4300 Wilson Blvd., Suite 750, Arlington, VA 22203, USA. E-mail: jmpaul@vt.edu
[2]ECE Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA. E-mail: bhm@ece.cmu.edu
[3]To whom correspondence should be addressed. E-mail: jmpaul@vt.edu

processing elements (PEs).(1–3) The applications these single chip heterogeneous multiprocessors (SCHMs) will execute will exhibit task-level heterogeneous concurrency, thus naturally motivating the use of multiple, heterogeneous PEs.

Many popular "divide and conquer" design strategies such as component-based design(4) and the orthogonalization of design concerns in platform-based design(5) make similar assumptions that performance may be improved in isolation. The value of specificity is measured in terms of relative gains without considering that specificity may introduce system-level losses in performance. For example, if the time to execute some algorithm, such as JPEG, contributes significantly to the overall execution time of the application, a system designer may seek to improve the overall execution latency of the application on a given design by improving the performance of JPEG executing in isolation. The isolation of design decisions has also motivated several efforts in application-specific processor (ASP) generation tools; a popular area of research today is focused on fast ASP synthesis.(6–10)

All of these approaches suffer from an assumption that improving the performance of one part of the system will not degrade the performance of another part. This basic assumption is supported by one of the few, fundamental laws in computing, Amdahl's Law,(11) which is taught in every undergraduate computer architecture course. Amdahl's Law is a truly elegant law that seems inviolate. However, it also contains some assumptions about the design space that fail to hold for SCHMs. As a result, fundamental assumptions about how performance is increased – or negatively impacted – in a design must be reconsidered. In this paper, we re-visit Amdahl's Law for SCHM designs. We motivate that performance improvement considered in isolation can not only have small performance impacts on the overall system (which was the original observation of Amdahl's Law), it can actually impede overall system performance. We observe that properties of heterogeneity and finiteness were not presumed to be factors when Amdahl's Law was first conceived. We conclude with some implications for the design of SCHMs.

2. SUMMARIZING AMDAHL'S LAW

The most fundamental of all speedup observations is Amdahl's Law(11) which states that the performance improvement realized by using a faster mode of execution is limited by the fraction of time the faster mode can be used.(12) We re-summarize Amdahl's Law here, so that we may revisit it later in this paper.

Suppose a system executes three tasks serially, T_1, T_2, and T_3, for a total execution time of 3. Suppose further that each task requires a computation time (latency) of 1 or $L(T_1) = L(T_2) = L(T_3) = 1$. Now suppose that this same series of tasks executes on a new system, where one of the tasks, T_3 has a performance improvement from $L(T_3) = L$ to $L(T_3) = L'$. Amdahl's Law states that the speedup of the system is limited by the serialized fraction of time any one of the tasks executes. Thus, the speedup S of the new system is limited to

$$S = \frac{1}{\left(1 - \frac{1}{3}\right) + \frac{1}{3}\frac{L'}{L}}.$$

This states that, in this case, as L'/L goes to zero, the absolute best speedup is $S = 1.5$. The idealized speedup can only be achieved if the execution of task T_3 can be completely done in parallel with the other tasks, its serialization effectively eliminated and therefore no longer contributing to overall system latency.

Another interpretation of Amdahl's Law is that there are diminishing returns for performance improvement of any one task (without loss of generality we refer to "serialized fractions" normally associated with Amdahl's Law as tasks). Thus, if the performance improvement of task T_3 is an order of magnitude, or $L'=L/10$, then the speedup of the system is only $S=1.42$. The performance benefit to the overall system is clearly much smaller than that of the task considered alone.

Amdahl's Law has been used widely in the analysis and design of computer architecture. It is normally used to point designers in the direction of where best to invest design effort and system resources. The clear implication is that focusing on resources that help improve performance over only a small portion of system execution can be wasteful. Another clear implication is that there exists a positive relationship between the performance improvement of any one task (or fraction) in the system and the overall system performance. When the performance of any one task in the system improves, the overall system performance will improve; however small that performance improvement might be, a positive relationship to overall performance is presumed to always exist.

This can be seen in Fig. 1, where three execution regions, r_1, r_2, and f, are shown in an original application with overall Latency, L_0. According to Amdahl's Law, the best a designer can hope for is to remove the time to execute the sequential fraction, f, by, for example, executing it in parallel with the other regions of the application, thereby completely removing its execution time. Thus, the best case, according to Amdahl's

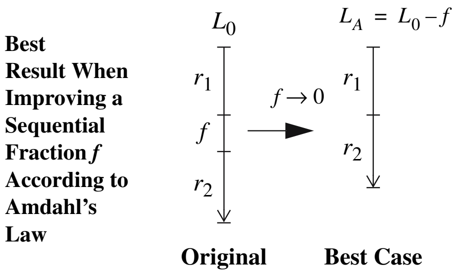
Best Result When Improving a Sequential Fraction f According to Amdahl's Law
L_0
$L_A = L_0 - f$
r_1
f
r_2
$f \to 0$
r_1
r_2
Original
Best Case

Fig. 1. Best result when improving a sequential fraction f according to Amdahl's Law.

Law is shown on the right, with overall latency of $L_A = L_0 - f$. The implication is that overall performance can only be improved by speeding up the time it takes to execute f, that the lowest latency one can achieve is L_A, and that, if the original time it takes to execute f was a small fraction of the time of the overall application, a designer's time might be better spent focusing elsewhere, where the dividends might be higher.

This conclusion, simple and powerful, contains two key assumptions, that the resources upon which the regions execute are:

1. unbounded and
2. homogeneous

These assumptions are more specific ways of examining one overall assumption of Amdahl's Law: that the performance improvements are presumed to be isolated from one another. However, when resources must be viewed as a trade-off within a bounded (finite) space, this assumption no longer holds.

Amdahl's Law was conceived in a day when parallel processing was presumed to take up an entire room, or even rooms, of computing space. The size of the computer had a logical boundary, but not necessarily a definite physical one. Whatever limited the actual size of a computer might vary. This is one reason scalability is such an important metric in parallel computing; scalability is targeted at identifying (and limiting) whatever impedes the performance benefits of systems as they grow in size. But the things that impede scalability can include communications bandwidth, topology, and even scheduling strategies. These are not, however, the overall size of the system, which is presumed to be able to grow (i.e., scale positively).

By contrast, SCHMs have definite, finite physical extent. There is a fixed boundary for the chip where the addition of one processor resource may take away from the capabilities of the resources on the rest of the chip. This is shown in Fig. 2.

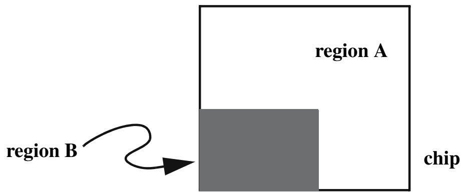

Fig. 2. Customization in a finite-sized system.

The chip as shown consists of two regions, region B, which is shaded, and region A, which is not. Region A is the "rest of the chip," i.e., everything else on the chip that is not part of Region B. Suppose region B has been customized to optimize a subset of the application, such as the sequential fraction, f, of Fig. 1. The real-estate on the chip dedicated to improve the performance of f may greatly improve the performance of f. However, what is the impact on the overall performance of the original application, including code regions r_1 and r_2, that must also execute on the same chip?

Motivated by this question, our own prior experimental results,(13), and some recent SCHM architectures, discussed in the next section, we examined this question in more detail.

3. MOTIVATION

Emerging single chip heterogeneous multiprocessors (SCHMs) might be more appropriately described as "processors of processors" as opposed to "processors of registers" or "processors of functional units (FUs)." These designs have organizing principles around which multiple instances of a class of single chip heterogeneous multiprocessor architectures might be designed, analogous to a family of microarchitectures. The design style of Systems-on-a-Chip (SoCs) has its origins in application specific integrated circuits (ASICs) and embedded system design, which tends to focus on the synthesis of a system for a given application (or set of applications). By contrast, SCHMs such as the Hyperprocessor,(14) the Cell,(15,16) and the Sandblaster(17) are more general hardware solutions, which focus on more general forms of programmability at the system (chip) level. Each has initially focused on a class of applications. The Hyperprocessor has focused on multimedia, the Cell on gaming, and the Sandblaster on software-defined radio. However, each architecture is also open to facilitating different classes of applications.

The Hyperprocessor removes the restriction that tasks are statically mapped to execute on resources most appropriate for the task type.

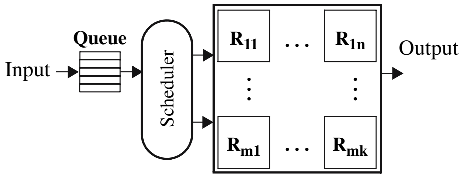

Fig. 3. SCHM with global task queue.

This opens up the possibility that system performance can be improved if tasks execute immediately on available resources instead of waiting for resources best suited to the task type. This situation is shown in Fig. 3. This hypothesis was experimentally verified in Ref. 13, which introduced Model-based Scheduling as a design strategy. Here a scheduler on each processing element in a SCHM is designed to the context of both a model of the performance capabilities of other elements in the system and the state of the system. Thus, the individual processing capabilities of the system, the current loading situation of the system, and the size and content of the datasets being processed by a set of tasks are all considered when processing tasks in a common task queue. The Cell and the Sandblaster can also utilize this more general approach to programming the system. Current research in the programming model for the Cell explores the use of a customized version of the Linux operating system, where scheduling decisions can also be made dynamically and at the chip level.

Amdahl's Law informs us that improvement in the performance of any resource of Ref. 3 will lead to overall system improvement; the only question is how much the resource is utilized over the execution period of interest. If this is not true, then Amdahl's Law cannot be used to inform designers of future SCHMs. Motivated by this question, we sought to investigate and illustrate it through a set of fundamental experiments, as well as propose some implications for designers of future SCHMs.

4. HETEROGENEOUS MULTIPROCESSOR SYSTEMS

Consider a SCHM where tasks are always mapped to the optimal resource, and stored in queues locally if the resource is not immediately available. Intuitively, in situations with aperiodic input behavior, the system may become load-imbalanced with some queues close to capacity while others are empty if performance is narrowly focused on the execution latency of individual tasks. Tasks of the same type are competing for a common resource, creating a bottleneck.

However, with a global task queue and global scheduler, the system of Fig. 3 can respond to loading situations by executing tasks on processor resources other than the one that provides the best performance for the task type. This affords more possibilities to respond to dynamic loading situations, since it is often better to schedule tasks that start sooner, but take longer to execute, than to wait for a resource that might provide faster execution time.(18,19) However, now the system-level effects of using a performance-tuned PE are more difficult to determine.

Motivated by how the Model-based Scheduling Example provided performance enhancement for a system like that of the Hyperprocessor and Fig. 3, we conducted further experiments on the Model-based Scheduling Example described in Section 3 in order to understand in more detail what lead to system speedup. In the following experiments, as well as in the Model-based Scheduling Example, tasks that execute on multiple resource types are presumed to be compiled for multiple resource types, i.e. multiple copies of the tasks are presumed to exist in the local, instruction memories of each processor that might execute each task type. Other methods might use dynamic binary translation. Either way there is overhead involved, one has impacts on space while the other has more impacts on execution time. We believe that the cost-benefit of introducing overhead to accommodate global design concerns in SCHMs is analogous to caches and branch predictors in microarchitecture – these are design elements that facilitate performance, but are really overhead that happen to facilitate overall performance. This is discussed further in Section 6.

4.1. Experiments

Our investigation begins by isolating how system speedup is affected by scheduling policies in specific situations.

We evaluated the total execution latency of *tasks* executing on a *SCHM* for a system structured like that in Fig.3. However, we fixed the input queue at three tasks and the number of processing elements in the system at two, so that we could isolate the latency of the processing time of the three tasks on the two heterogeneous processor resources. The example system is shown in Fig. 4. We define the experimental parameters of the system as:

- $tasks = \{T_1, T_2, T_3\}, T_i \in$ {gzip, wavelet, quantization},
- $HM = \{R_1, R_2, X\}, R_i \in$ {ARM, DSP},
- $X \in \{X_1, X_2\}$, global, dynamic schedulers defined later,

where HM is an abbreviation for Heterogeneous Multiprocessor.

In this system three tasks, T_1, T_2, T_3, appear in order in the input queue of the system of Fig. 4. They are scheduled onto two processor

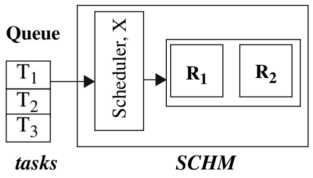

Fig. 4. Example system.

resources, R_1, R_2, by a global dynamic scheduler, X, which is one of two scheduler types (discussed later). T_i is one of the possible task types, which in this case are gzip text compression, wavelet transform and zerotree quantization. Each task in the input queue can be any of the three task types, and all possible permutations are considered. R_1 and R_2 are the processors, an ARM and a digital signal processor (DSP) for this example. The schedulers are aware of only the first item in the queue and the state of the processors. The relative performance differences of each of the three tasks on the two processors is summarized in Table I.

Maximum system latency *L(tasks, HM)*, is how long it takes for all tasks to complete. Tasks are scheduled by order of appearance in the queue, but otherwise have no dependencies. A system, *SCHM*, becomes a different system, $SCHM'$, when any of R_1, R_2, or X are altered. *L(tasks, HM)* was evaluated using two global dynamic scheduling policies, X_1 and X_2.

The two global dynamic schedulers, X_1, X_2, used in our experiments differ in how they determine on which processor T_3 should be scheduled. Both schedulers schedule tasks T_1 and T_2 on the best available resource by order of appearance in the queue. This translates to T_1 being scheduled on the optimal processor for its task type. T_2 is then scheduled on the other processor since it is the only processor available. X_1 then schedules T_3 on the first resource to become available (since both are busy when T_3 is considered for scheduling), while X_2 schedules T_3 on the better performing resource for the task type of T_3.

Table I. Relative Task Performance

	gzip	wavelet	quantization
ARM	2	0.5	2
DSP	1	1	2

We specifically constructed the example this way in order to isolate the effects of serialization and speedup in SCHM systems. Speedup is normally considered when there is some serialization present. If all tasks can execute in any order, then speedup is trivial to consider. In this case, the serialization is the order of tasks in a queue. The first two tasks in the queue must be started before the third. Thus the third task has a simple serial dependency, but since our objective is to distill an observation, this simple serialization is adequate for our purposes. Our goal is to isolate the effects of scheduling decisions on task *types*, given some ordering of the system that provides a baseline loading condition (the scheduling of the first two tasks) and a decision to be made about how task type can be dynamically scheduled onto processor type, considering the capabilities of the processor resources and the system loading encountered when the scheduling decision is made.

In our experiments, we found that when $T_1 = \text{wavelet}$, $T_2 = \text{wavelet}$, $T_3 = \text{gzip}$ and the scheduling policy is changed from X_2 to X_1, the system latency improves even though the performance of T_3 is individually degraded. This is illustrated in Fig. 5.

In this case T_1 is scheduled on the ARM, and T_2 on the DSP, which means that each processor in the system is processing a wavelet job. However, because the DSP processes the wavelet jobs twice as fast as the ARM, the DSP becomes available well before the ARM becomes available. Thus, even though the processing time of the gzip job remaining in the queue, T_3, would be twice as fast on the ARM as it would be on the DSP, it is more beneficial to overall system latency to schedule the gzip task sooner and process it slower by scheduling it onto the DSP, rather than wait for the resource for which it is more suited, the ARM.

While the observation that execution latency can be improved by degrading the performance of a particular task is not an intuitive result, it has been observed as a scheduling anomaly on homogeneous processors.(18) However, in our example, we considered the effect not as an anomaly – an undesirable result – but a design trade-off. This observation

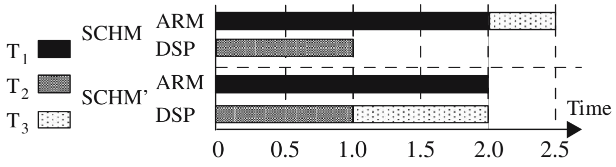

Fig. 5. ARM/DSP timing diagram.

prompted us to re-visit the concept of speedup, which we then use to re-visit the use of application specific processors in SCHMs.

We have illustrated that this can occur when heterogeneous tasks share heterogeneous resources as in systems of the type of Fig. 3, and that performance improvement of any one task cannot be isolated. Speedup is a trade-off and cannot be considered in isolation.

While this effect is not unique to SCHMs, this effect is more exposed in SCHMs because of the coarse granularity of tasks and processors as opposed to instructions, functional units and registers in conventional processors. It is additionally exposed for SCHMs because performance has a tendency to be evaluated more as latency for specific applications as opposed to throughput over a number of instruction (or in the case of SCHMs task) types.

This is because in many SCHM applications, such as multimedia, games, and software-defined radio, applications *persist* in the system over a long period of time (which is a common assumption in embedded system design), but the system are also *multi-modal*, i.e., different applications may exist in the system at different points in time.

5. AMDAHL'S LAW REVISITED

Amdahl's Law suggests that design decisions that speed up the execution of any task, or execution region, leads to systems-level speedup (however small). This is illustrated once again, in Fig. 6, which is Fig. 1 re-drawn to include what we observed.

In Fig. 6, once again, the leftmost line represents the critical execution path of a system with serialized execution latency L_0. In this system, r_1 and r_2 denote the latency of regions surrounding a third region with latency f. The region with latency f represents some sequential fraction of execution targeted for performance improvement.

We observed that, as the contribution of f to the total system latency goes to zero, system latency behaves in one of two ways. In case A, the

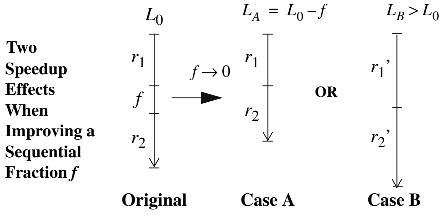

Fig. 6. Two speedup effects when improving a sequential fraction f.

system latency is $L_A = r_1 + r_2$ and the system speedup is

$$S = \frac{r_1 + f + r_2}{r_1 + r_2}.$$

Speedup is greater than 1 and overall system performance is improved. This is Amdahl's Law. The performance improvement has been made to f either completely independent of r_1 and r_2 or with the assumption that any change in the design that contributed to the performance improvement of f produced negligible effects on the performance of regions r_1 and r_2.

In contrast, case B shows the effect we observed experimentally, first by our work in Ref. 13 and again in the much simpler, distilled example in this paper.

In case B, by improving the performance of the fraction f we observed that performance in regions r_1 and r_2 (or both) were affected adversely enough so that total latency of the system was actually made worse by improving the execution time of f.

Stated another way, even though performance of the region f was improved, the effect that led to the performance improvement of f caused $r_1' + r_2' > r_1 + f + r_2$, so that $S < 1$ for case B. Overall system performance has been degraded instead of improved because of side effects introduced by improving the performance of f.

Significantly, this observation is not limited to the case when heterogeneous tasks execute on heterogeneous resource types. In any system, the improvement of the execution of sequential fraction, f, can come at such a price that the execution of regions r_1 and r_2 is adversely affected. This is analogous to an extremely large Region B in Fig. 2, at the expense of smaller chip real-estate left over to process regions r_1 and r_2.

As a further illustration, consider a uniprocessor system that lacks a floating point unit, but spends 50% of the time emulating floating point. Adding a floating point unit would likely benefit the overall system speedup. However, if the addition of the floating point unit causes other architectural features of the processor to be removed or made smaller or less complex, such as the caches, register files, and branch predictors, then the performance improvement provided by the floating point unit may well be more than offset by the performance degradation when the system is executing something other than floating point operations. The addition of the floating point unit in this case must be considered with respect to more than just its utilization and the diminishing returns it may provide to overall system speedup, i.e., more than what Amdahl's Law tells us. In this case, it represents a trade-off where speeding up one part of the system may slow down the other parts to the point that a local speedup does not reflect a more global, or system-level speedup.

The simple microarchitecture example suggests that the failure of Amdahl's Law to capture finite systems is an observation that has been with us for some time. And yet, Amdahl's Law continues to be taught, without qualification, and its basic assumptions continue to be used to justify significant research efforts that, instead of considering global, system-wide effects, focus on "divide and conquer" component views.

6. IMPLICATIONS

With new fundamental models come implications for designers. In this section, we focus on three of what we feel are the most significant ones. The first focuses on the impact of processor specificity. Considerable research effort is devoted to fast generation of custom processors. We attempt to shed light on the relative merits of fast, custom processor generation vs. focus on system-level design. The second examines the impacts of custom chip-level schedules that can take advantage of heterogeneous processor resources, even if the processor resource is not ideally suited to a given task. The third examines the implications of chip-level partitioning, away from side-by-side partitioning towards global-local forms of partitioning.

6.1. Processor Specificity on an SCHM

The previous illustration focused on the impact of scheduling policy given a set of heterogeneous PEs. However, designers of SCHMs will also have the opportunity to select the numbers and types of PEs on a chip. The question in this case is: given a set of tasks and a scheduling policy, what is the best set of heterogeneous processors for a system? Motivated by our previous observation, it seems intuitive that processor specificity could sometimes have an adverse impact on overall system speedup for a chip of fixed area; different processor types might either create an access bottleneck, or be a poorer performer than some other processor on tasks the system might execute more often.

When we set out to examine the effects of processor specificity within an SCHM, we found no models that relate task type, processor type, and performance. And yet intuitively, DSPs perform better on DSP task types because DSP task types include more instructions or instruction patterns that execute more efficiently on a DSP, with the implication that those same patterns execute more poorly on a more general purpose processor (GPP). While many processor designers think in terms of broad categories of task types, heterogeneous multiprocessor designers must be aware of finer-grained differences in task types, and the corresponding relationship with processor type. We postulate what such relationships might look like and why. We begin by developing a processor classification given a set of tasks. To make such comparisons fair, processor types are presumed to have made sacrifices for their specificity – DSPs, for example, may have given up sophisticated branch prediction in exchange for a variety of advanced arithmetic operations.

Our classification in this section has been based on intuition: we do not claim to have derived it experimentally. We realize that the relationships we assert will be different given a different sets of tasks. Our point is only that it should be possible to order tasks by type where a fixed amount of complexity (instructions, lines of code or some other measure) results in different performance on different processor types. This is required in order to have a meaningful discussion of heterogeneity and the role application specificity plays in the broader vision of SCHMs and other heterogeneous multiprocessors, since it allows us to compare processor types of varying specificity.

6.1.1. Relating Task and Processor Types

We presume we can order a set of tasks $\{T_0, T_1, \ldots, T_n\}$ by similarity of type (similarity may be defined in a number of ways – e.g., quantity of memory accesses, amount of times a certain operation used). Tasks close to one another in the type sequence exhibit similar performance on a given PE type. This seems reasonable since all tasks can be broken down into instructions, which can be counted and ordered by similarity. Next, we classify PEs in four broad categories of: GPPs, custom hardware (CH), ASPs, and class-specific processors (CSPs). Note that CSPs are more commonly referred to as domain-specific processors, but we use the name class-specific to differentiate our abbreviation, CSP, from that of DSPs. Then, we build a set of relationships between task type and processor type according to their *normalized performance*. The normalized performance of a PE executing a task T_0 is the rate at which it is able to execute units of complexity of task T_0 relative to a GPP (GPPs have constant normalized performance).

In GPP design, effort is only invested in architectural features that can accelerate general execution. Since GPPs are (ideally) designed to execute all tasks equally well, we show the normalized performance of GPPs as a constant. CH, by contrast, is specifically designed to compute a single task and no others; CH is not a generally programmable device. The normalized performance of CH is thus undefined for all tasks but the one for which it is designed, but CH typically performs better than any other PE for the specified task, in this case, T_0.

We consider ASPs a hybrid between GPPs and CH. ASPs are programmable (Turing complete) processors with special purpose hardware integrated into their datapaths. As a result they are designed to execute a narrow class of tasks especially well. For example, motion estimation in MPEG might be executed as a single, custom instruction in an ASP. An ASP performs best when executing task T_0, which could be conceived of as the custom hardware being repeatedly accessed in a tight loop with no other instructions executed by the processor (e.g. constantly executing motion estimation). The peak performance of ASPs tapers off when executing tasks that make less use of its custom hardware or instruction. The performance of tasks "far away" from T_0 may actually exhibit poorer performance than if they were executed on a GPP; this makes intuitive sense, since there is at least some overhead associated with the extra hardware that is not being utilized by tasks far away from T_0.

We define CSPs as processors that provide performance improvement over that of a GPP for a broader range of tasks than an ASP. Probably the most common CSPs are DSPs, where features like multiply-accumulate units (MACs), floating point units (FPUs), vector operations and support for complex arithmetic provide support for a broad class of applications that can take advantage of these features. Because compilers tend to produce patterns of instructions, the performance curve for CSPs is presumably flatter than for an ASP, where the special hardware is often accessed by hand insertion of a call.

The normalized performance of all PE types for a set of tasks $\{T_0, T_1, \ldots, T_n\}$ is summarized in Fig. 7. In the figure, the right-most task, T_0, orients the shape and size of the set of curves.

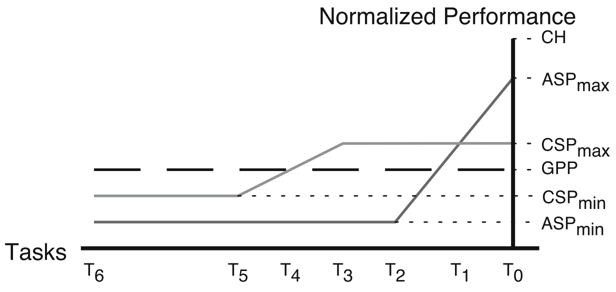

Fig. 7. PE performance summary.

6.1.2. Performance Operating Regions

In order to explore the impact processor specificity has when the SCHM executes multiple task types of varying similarity on multiple processor types, we divided Fig. 7 into six operating regions $\{A_1, A_2, A_3, A_4, A_5, A_6\}$ as shown in Fig. 8.

Regions A_5, and A_6 are marked by the increased relative proficiency of GPPs, which is intuitive under the assumption of constant design cost (e.g., area, complexity). CSPs and ASPs make use of hardware not present in a GPP to implement special functionality. Given our concept of normalized performance, the same real-estate would be used on a GPP to accelerate control flow with logic dedicated to branch predictors, trace caches and the like which improves the performance of all tasks. For some tasks, the gains from special functionality in ASPs and CSPs will be overcome by the GPP.

When comparing real processors, the boundaries of GPP/CSP/ASP are quite fluid. However, the shape of regions A_1, A_2, and A_3 is at least consistent with the performance of real ASPs and CSPs. In Ref. 20 an application specific DSP (ASDSP) is discussed and tuned to perform FFT operations. It excels when compared with two general DSPs (30% average improvement). Since the proposed architecture actually reduces the area needed for its data processing unit relative to a comparison DSP, it is reasonable to conclude that the ASDSP may perform worse than comparison DSPs for some other set of DSP tasks.

Next consider a set of PEs executing $\text{Tele}_{\text{mark}}$, a benchmark in the Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark suite.(21) A BOPS Manta v.2, a DSP with indirect VLIW, performs 70% better than a IBM 405GPr-200, a GPP, only on a subset of autocorrelation tasks and 60% worse otherwise, making it an ASP over the $\text{Tele}_{\text{mark}}$

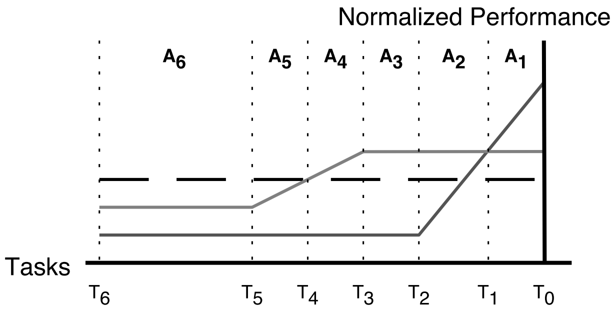

Fig. 8. Operating regions for processor classes.

kernels. On the other hand, an Infineon Carmel 10xx-170, a DSP with CLIW, outperforms the GPP on all autocorrelation tasks by an average of 40%, and on a subset of fixed-point bit allocation tasks by 500%, but 32% worse otherwise. This makes it application specific over fixed-point bit allocation but class-specific over autocorrelation.

6.1.3. Illustration

Using the regions of Fig. 8, we ran a set of experiments to examine the potential impact of processor type specificity on overall system speedup. As in Section 4, we define the experimental parameters of the system as:

- $tasks = \{T_1, T_2, T_3\}, T_i \in \{A_1, A_2, A_3, A_4, A_5, A_6\}$, the operating regions defined and discussed in Section 6.1.2
- $HM = \{R_1, R_2, R_3\}, R_i \in$ {ASP, CSP, GPP}, the processor types defined and discussed in Section 6.1.1
- $X \in \{X_1, X_2\}$, the schedulers discussed in Section 4.

The relative processor power for the center of the operating regions of Fig. 8 used in our experimentation is in Table II. While these are arbitrarily assigned they are consistent with reasonable performance relationships for a given task type.

Note in particular that the ASP is an order of magnitude better than the CSP while executing in region A_1.

As before, three tasks are scheduled onto two processor resources and the system latency is measured. However, in these experiments PEs are varied instead of the system scheduling the types. Figure 9 illustrates one example of our experimental findings.

For $SCHM$ in Fig. 9, $T_1 = A_3, T_2 = A_2, T_3 = A_1, X$ is X_1. T_1 and T_2 are first scheduled, one on each ASP. Since the scheduling policy for $SCHM$ is X_1, T_3 is scheduled on the first available PE. Because T_1 is an A_3 task, it takes longer to execute on an ASP than T_2 and T_3

Table II. Relative Performance

	Region					
PE type	A_1	A_2	A_3	A_4	A_5	A_6
ASP	10	1	0.5	0.5	0.5	0.5
CSP	2	2	2	1.5	0.87	0.75
Gpp	1	1	1	1	1	1

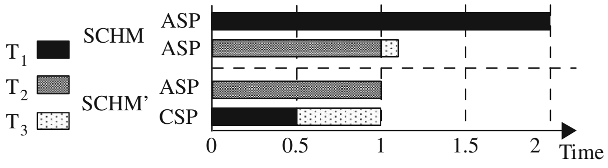

Fig. 9. Example timing diagram.

combined; the system latency for SCHM is 2, and the latency of T3 is 0.1. *SCHM′* uses a CSP in place of one of the two ASPs. When T_1 is scheduled, the CSP is the best available processor; T_2 is scheduled on the ASP. T_3 is again scheduled on the first available processor, which in this case is the CSP, which executes T_3 more slowly than an ASP. Despite this, the system latency of $SCHM'$ is improved over $SCHM$ at 1. The latency of T_3 is 0.5.

The presence of the ASP improved the performance of task T_3 at the expense of overall system performance by trading off the performance of T_3 for the performance of T_1. A more general class of processor improved overall system performance. The impact of believing that individual tasks executing on individual resources may be optimized in isolation is clear: by focusing performance improvement in part of the SCHM, making a custom processor that improves part of the task set but hinders some other, overall performance can suffer. Instead, more general processors might be more useful, when used with global scheduling techniques.

6.2. Impacts of Global Scheduling

We summarize the example used in Ref. 13 because it motivated a more detailed consideration of the design principles that lead to this paper, as well as provides some insight as to the impact of global scheduling. The system is a simple "multimedia" system consisting of two heterogeneous processors (a Renesas M32R, hereafter referred to as a DSP, and an ARM) to handle a mixture of image and text compression applications. The image compression algorithm is composed of two sequential tasks: a wavelet transform and zerotree quantization. The wavelet step transforms the image into a series of frequency sub-bands using digital filter techniques. Zerotree quantization then selects data to discard using control intensive algorithms. Text compression is performed using the gzip program that looks for redundant data patterns in plaintext, mainly stressing memory bandwidth and latency.

Each processor in this system was observed to execute the three tasks at different rates relative to one another. The DSP has a MAC instruction that makes it better suited for performing wavelet transforms. While both processors are equally adept at quantization, the ARM is approximately twice as fast at gzip text compression and the DSP is approximately twice as fast at the wavelet transform.

Processing inputs from a testbench that streamed compression packets at a Poisson rate, in order to explore a wide variety of data-dependent effects, a model-based scheduler was able to provide the best overall system performance by intelligently scheduling jobs onto “inappropriate” resources sooner rather than wait for the optimal resource when loading became too severe (i.e. too many wavelet tasks waiting for the DSP). It was possible to improve system performance by taking advantage of more situations where individual tasks would actually execute slower on the processing resource least suited to its task type. It was also possible to introduce the overhead of gathering system information, including state about the system resources, utilize a more complex scheduling algorithm, and execute tasks on “inappropriate” resources, and still experience overall performance gains. The benefits of introducing the overhead of system-level coordination have analogies to microarchitecture, where caches and branch predictors introduce overhead that winds up improving the overall performance of this processor.

This points in the direction of the importance and impact of system, i.e., chip-level, global decision making. The clear implication is that more research efforts should go into the hardware and software support for minimization and optimization of information upon which global decisions are made. This is, potentially, a more fruitful area than one in which isolated performance regions are sped up with custom resources.

6.3. Global-Local Partitioning

The previous sections show that custom performance tuning of programmable processing elements to applications, without consideration of how they impact the overall performance of a chip, may negatively impact overall performance. With increases in overall complexity on all systems comes the likelihood that systems will exhibit multiple modes of operation, including modes that cannot take advantage of the performance specialization of some custom processors. For example, a cell phone is often used in ways that have nothing to do with actually making a call. Other modes of operation for the same device require different resources. Many design tools and much research effort is expended towards the goals of divide and conquer and the generation of custom design elements. Amdahl’s Law supports this widely held view that divide and conquer design strategies are adequate ways to handle performance. This results in what might be thought of as a horizontal, or side-by-side view of partitioning of a chip. However, horizontal partitioning fails to capture global design concerns.

To point in the direction of new ways of thinking about system partitioning, we include a simple illustration of the existence of two kinds of state in a single chip. The first kind of state is that which is grouped into horizontal regions conventionally associated with divide and conquer partitions. This is conventional state that we will refer to as "computation state" since it is normally associated with the carrying out of some specific functionality. The second kind of state is which is global to all regions of the chip. Since global state might be thought of as providing a layer across all elements of the chip, it might be thought of as a logical vertical partition. The most natural way to think of state that belongs to a global region is system state, since state in a global region is likely to have benefit only if it is used to coordinate activities across multiple regions of the chip.

Figure 2 shows the partitioning of a chip into two regions, one presumed to benefit one kind of application and the remainder of the chip. This kind of partitioning is typical of many hardware designs where the emphasis is placed upon individual components and their integration. This can be thought of as horizontal, or side-by-side partitioning, which creates local domains (such as processing elements) on the chip. However, our observations suggest that the horizontal, or local-local, partitioning associated with hardware design is insufficient for single chip, finite systems where global decision-making is important.

By contrast, Fig. 10 shows each chip partition divided into two kinds of state, where the *ci* represent computation state and the *si* represent system state. Since system state is state around which chip-level decision making can be made it must be considered global to the chip. Figure 10 is shown partitioned into nine computation blocks, indicated by the squares with the [*ci,si*] pairs as labels, divided by a diagonal line on each block. Classical, horizontal, or local-local, partitioning considers the division of the chip into partitions by space and the kind of labor the chip does. While the partitions are shown as identically-sized for the sake of simplicity, it may be that block $[cl, sl]$ takes up a large region of the chip. Thus, region B of Fig. 4 might correspond to block $[cl, sl]$ while all of the other blocks might correspond to region A of Fig. 4.

With increased complexity on a given chip comes the ability to execute applications with increased complexity and heterogeneity. On many single chip systems each region is a programmable processing element that while considered more optimal for one type of application vs. another is still Turing complete. This means that any task is potentially capable

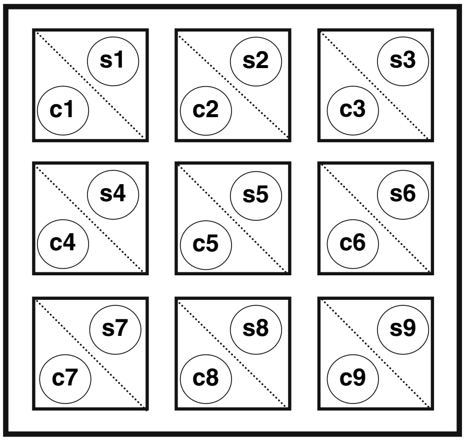

Fig. 10. Illustration of global-local partitioning.

of executing anywhere. A chief concern then is the utility of any given region on the chip. Most applications will persist in future systems over long periods of time (unlike batch jobs of days of old). Most systems will also be multi-modal, where few, if any, applications persist in the system 100% of the time. Thus, each resource in the system can be programmed to take on one or more applications, potentially even dynamically. The cost of reconfiguring the system can be considered the cost of implementing and making decisions about system state, which is state that is treated as global state, even if it is logically distributed on the chip.

We propose the need to consider two forms of partitioning of a chip: (1) local–local partitioning, or the partitioning of the chip into the *ci* blocks in the first place, where local applications are computed and (2) global–local partitioning, or the partitioning of each of those *ci* blocks into a contribution to overall, chip-level system state, the *si*. Examples of the benefits use of system state to make decisions can be found in Ref. 22. Tools and design strategies must support the cost-benefit analysis of global-local partitioning.

7. OBSERVATIONS AND CONCLUSIONS

Amdahl's Law is based upon two assumptions – that of boundlessness and homogeneity – and so it can fail when applied to single chip heterogeneous multiprocessor designs, and even microarchitecture, as well as other computer systems.

We re-examined the implications of Amdahl's Law on SCHM designs, by re-examining the assumptions upon which it is based. By showing the limitations and assumptions inherent in Amdahl's Law, we motivate the need to reconsider research and design efforts that focus on divide and conquer. Instead, we advocate the need to develop tools, techniques, researcher and designers that can support holistic evaluations.

We realize that Amdahl's Law is one of the few fundamental laws of computing. However, its very power is in its simplicity, and if that simplicity is carried over to future systems, we believe that it will impede the potential of future computing systems.

We showed that properties of heterogeneity and finiteness that are present in SCHMs are not a part of the assumptions upon which Amdahl's Law is based. Ignoring these assumptions can hinder performance-optimal design for SCHM systems and even microarchitecture as well as other computer systems; designers can speed up one part of the system, one task or program fragment, and inadvertently slow down the system overall.

We motivate that research and design effort would be better focused on processors that provide performance improvement for a broader set of task types rather than the more narrow performance improvement targeted by application specific processors. It may also be more advantageous to invest effort in developing more sophisticated system-aware scheduling rather than in using off the shelf schedulers and operating systems. Finally, design strategies that partition in a global–local manner, rather than a component-based, side-by-side manner should be pursued far more than they are now – and away from current emphasis on component-based design and the isolation of design concerns.

Fundamental assumptions that are consistent with Amdahl's Law are a heavily ingrained part of our computing design culture. This paper points in a new direction. New tools and design strategies are required that support holistic views, instead of divide and conquer approaches.

ACKNOWLEDGMENTS

This work was supported in part by ST Microelectronics, and the National Science Foundation under Grants 0607934, 0606675, 0438948, and 0406384. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The authors thank Don Thomas, Alex Bobrek, and Sean Pieper for their feedback and suggestions. We also thank the reviewers for seeing the contribution of the paper and for their suggestions.

REFERENCES

1. Are Single-Chip Multiprocessors in Reach? *IEEE Design and Test*, **18**(1):82–89 (Jan–Feb 2001).
2. W. Wolf, How Many System Architectures? *IEEE Computer*, **36**(3):93–95 (March 2003).
3. T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, and W. Wolf, Mobile Supercomputers. *IEEE Computer,* **37**(5):81–83 (May 2004).
4. W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, Component-Based Design Approach for Multicore SoCs, *DAC*, *Proceedings of the 39th Design Automation Conference*, New Orleans, LA, pp. 789–794 (June 2002).
5. A. Sangiovanni-Vincentelli and G. Martin, Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design and Test*, **18**(6):23–33 (Nov–Dec 2001).
6. C. Rowen, *Engineering the Complex SoC*, Prentice Hall (2004).
7. D. Goodwin and D. Petkov, Microprocessor Architecture: Automatic Generation of Application Specific Processors, *CASES*, *Proceedings of the 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, San Jose, CA, pp. 137–147 (October 2003).
8. A. Mihal, C. Kulkami, M. Moskewicz, M. Tsai, N. Shah, S. Weber, J. Yujia, K. Keutzer, K. Vissers, C. Sauer, and S. Malik, Developing Architectural Platforms: A Disciplined Approach, *IEEE Design and Test*, **19**(6):6–16 (Nov–Dec 2002).
9. N. Clack, H. Zhong, and S. Mahlke, Processor Acceleration Through Automated Instruction Set Customization, *Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, CA, pp. 129–140 (December 2003).
10. Tensilica Inc., Xtensa Product Brief, http://www.tensilica.com (2002).
11. G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proc. AFIPS Spring Joint Computer Conf. 30*, Atlantic City, NJ **30**: 483–485 (April 1967)
12. J. L. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd Ed., Morgan Kaufmann, pp. 40–41 (2003).
13. J. Paul, A. Bobrek, J. Nelson J. Pieper, and D. Thomas, Schedulers as Model-based Design Elements in Programmable Heterogeneous Multiprocessors, *DAC*, *Proceedings of the 41st Design Automation Conference*, San Diego, CA, pp. 287–292 (June 2004).
14. F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, A Multilevel Computing Architecture for Embedded Multimedia Applications, *IEEE Micro*, **24**(3): 56–66 (May–June 2004).
15. P. Hofstee and M. Day, Hardware and Software Architectures for the Cell Processor, *Proceedings of the 3rd International Conference on Hardware/Software Codesign and System Synthesis*, Jersey City, NJ, pp. 19–21 (September 2005).
16. D. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa, Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor, *IEEE Journal of Solid State Circuits*, **41**(1):179–196 (2006).
17. M. J. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis, A Low-Power Multithreaded Processor for Software Defined Radio, *Journal of VLSI Signal Processing*, **41**:143–159 (2006).

18. J. Madsen, K. Virk and M. J. Gonzales, A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor Systems-on-Chip, in *Multiprocessor Systems-on-Chip*, A. Jerraya and W. Wolf eds., Morgan Kaufmann (2004).
19. G. Buttazzo, Achieving Scalability in Real-Time Systems, *IEEE Computer*, **39**(5): 54–59 (May 2006).
20. K. L. Heo, S. M. Cho, J. H. Lee and M. H. Sunwoo, Applcation-Specific DSP Architecture for Fast Fourier Transform, ASAP, *Proceedings of the 14th International Conference on Application-Specific Systems, Architectures, and Processors*, The Hauge, The Netherlands, pp. 369–377 (June 2003).
21. Embedded Microprocessor Benchmark Consortium, http://www.eembc.org
22. J. M. Paul, Donald E. Thomas and Alex Bobrek, Scenario-Oriented Design for Single-Chip Heterogeneous Multiprocessors, *IEEE Transactions on VLSI*, **14**(8):868–880 (August 2006).