Tuning SoCs using the Global Dynamic Critical Path

Hari Kannan[1] Mihai Budiu[2] John D. Davis[2] Girish Venkataramani[3]

[1] *Computer Systems Lab, Stanford University* [2] *Microsoft Research, Silicon Valley* [3] *Mathworks, Inc.*

ABSTRACT

We propose using a profiling-based technique (Dynamic Critical Path) to guide SoC optimization. Optimizing SoCs composed of many modules involves exploring a large space of possible configurations (exponential in the number of component modules). We present this optimization technique applied to a Globally Asynchronous Locally Synchronous (GALS) RTL design. Furthermore, we investigate the loss of precision when abstract versions of hardware modules are used for the critical path computation. Using the critical path provides very fast convergence towards optimal or near-optimal solutions when analyzing large configuration spaces by optimizing the design for composite optimization metrics, such as energy-delay.

I. INTRODUCTION

Contemporary SoCs are composed of tens or even hundreds of IP blocks, each of which can be independently tuned, giving rise to a huge space of possible configurations. The blocks used by manufacturers in their designs may come from a variety of internal and external sources. Regardless of the SoC IP block source, the internal operation of modules and associated corner cases may not be well understood, or transparent to the SoC designers. This is further complicated by the fact that third party vendors of IP blocks do not provide source code access for their modules. All of these factors make performance analysis and optimization of SoCs or Multi-Processor SoCs (MPSoCs) extremely difficult.

Recent work has established dynamic critical path (or global critical path, GCP) analysis [6] [13] as a powerful tool for understanding and optimizing the performance of highly concurrent hardware-software systems. The GCP provides valuable insight into the control-path behavior of complete systems, and helps identify bottlenecks. It tracks the chaining of transitions of the key control signals and identifies the modules or IP blocks that contribute significantly to the end-to-end computation delay.

In this paper, we propose using the GCP to identify and remove system-wide bottlenecks in MPSoCs. Using this knowledge, designers can better direct their optimizations: to boost the performance of underperforming modules, lower power consumption, reduce excessive resources, etc. In the absence of such a tool, designers are often forced to simulate many combinations of the various possible configurations [4] in order to arrive at an optimal design. We use the GCP analysis to drive a directed search to allow designers to efficiently explore the search space for configuration parameters, arriving at optimal or near-optimal configurations much faster than exhaustive searches. Using a power-delay product as the exemplar cost function, our algorithm efficiently discovers the optimal combination of parameters for the IP blocks that constitute the SoC design. Alternative SoC optimization techniques based on numerical design optimization, such as simulated annealing [7], or evolutionary algorithms [5] require significant simulation time, especially for large designs. This problem is exacerbated by the lack of intuition about the functioning of unfamiliar (often third-party) IP blocks, and by the extremely large search space, exponential in the number of IP blocks.

II. Global Critical Path Definition

The formal definition of the Critical Path in operations research is "*the longest path in a weighted acyclic graph*" [13]. An informal notion of critical path has been used for a long time at various levels of system views, including asynchronous circuits [3], modeled as Petri nets [15] and synchronous circuits [6][11], as well as software modules [12], network protocols [1] and multi-tier web services. The critical path is also related to critical cycles in pipelined processors [2]. The GCP should not be confused with the traditional notion of static critical path in synchronous circuits, which is defined to be the longest of the possible signal propagation delays between two clocked latches. In contrast, the dynamic GCP is related to the concept of instructions per cycle (IPC) for processors, since it is dependent on a particular workload (which is why the path is called "dynamic").

Modeling a hardware circuit as a graph [14], the nodes in the graph are functional units and the edges are signals, shown in the rounded box in Figure 1. To define the GCP, we have to consider an execution of the circuit, for a particular input; then we "unroll" the execution of the circuit producing a timed graph. Each relevant time moment contains a replica of the entire circuit as shown in Figure 1.

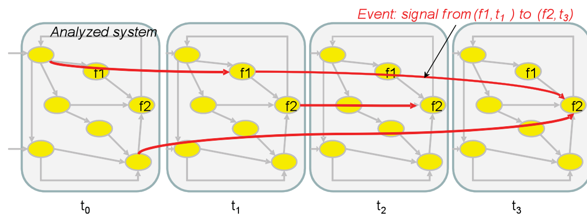


Figure 1: The Global Critical path is the longest chain of events in the timed graph.

The edges of the timed graph are *signal transitions*: an edge between (f1, t1) and (f2, t3) represents a signal leaving functional unit f1 at time t1 and reaching f2 at time t3. Edges from a functional unit to itself such as (f2, t1) to (f2, t2) represent computation delay. The longest chain of events in the timed graph is the GCP. Normally, only control signals need to be considered as parts of the GCP, because data signal transitions do not influence the timing of outputs.

III. Applying GCP to SOCs

GCP is easy to compute for asynchronous circuits because all signal transitions are explicit. Applying GCP to synchronous circuits presents many challenges that we address in this section. In particular, we discuss how GCP can be applied in practice for analyzing SoC designs with the added complexity of multiple clock domains.

A. Computing the GCP

The key idea for computing the GCP over all the modules is to track *dependencies between control signals.* We rely on an algorithm proposed in [6] for computing the GCP. For each module, we track the input and dependent output transitions. Whenever an output signal makes a transition (i.e., the module produces a new output value), we must be able to attribute it to a previous input transition, which triggered the computation. We only consider the *last arrival input* that caused this output (an output may depend on multiple inputs). We can construct the GCP using only local module data by stitching the local transitions, starting with the last transition of the system, and going back to the last arrival input which caused that transition. Recursively, this last arrival input becomes the last transition, and the algorithm is repeated until the start state is reached. This chain of edges is the GCP. The GCP is usually a large data structure, so we represent the GCP compactly as a *signal histogram*: for each signal of the circuit we count how many times its transition appears on the critical path. A signal with a high count is more critical than one with a low count.

B. GCP Accuracy

We are interested in understanding the loss of fidelity that can occur by using approximate RTL models of the hardware. The GCP computed using the lowest level RTL is the *ground truth* GCP; the GCP computed using abstracted models is just an approximation. Because we apply the critical path analysis to the RTL design, we have the flexibility of examining the critical path at a variety of levels: within the modules, at the module interfaces, or higher. We could also apply the GCP to abstracted views of the design such as electronic-system-level (ESL) models, or transaction-level models [7], but GCP fidelity is a concern (since these models are not always derived from the underlying RTL).

Given our definition of the GCP, there are three requirements for a model to produce an accurate estimate of the GCP: (1) it must model all concurrent hardware blocks, (2) for each hardware block, it must model the correct dependencies between input and output control signals, and (3) it must model transaction interleaving in the correct order (e.g., the arrival ordering of two input signals should not be swapped).

C. GCP for Synchronous Circuits

Unfortunately, applying this methodology to synchronous RTL-level circuits is not entirely straightforward. The GCP is very easy to build for handshake-based asynchronous circuits, because *all signal transitions are explicit* – and the critical path is composed of signal transitions. In clocked circuits the computation of the GCP has to account for the following issues, which we discuss in more depth in a companion technical report [9]:

1. I/O dependences in control FSMs are implicit.
2. Don't cares in control logic cause false I/O dependences.
3. Events that occur in the same clock cycle can be ordered in several ways.
4. Implicit signal transitions: some signals never change value, but still imply two transitions.

5. Inter-module handshake is sometimes implicit, such as in the presence of global stall signals.
6. Pure sources and sinks in the dependency graph create a circuit graph that is not strongly connected.
7. Signals with fanout: a signal with fanout may be last arrival only for some of the consumers.
8. Modules with multiple outputs: the criticality of each output must be considered separately.
9. Systems with non-deterministic inputs (interrupts): the hardware path exercised by the interrupt response will be the critical path. Since these inputs are by definition non-deterministic, no two runs of the global critical path will provide the same results.

IV. Evaluation System

We use an SoC composed of 6 modules that can be independently optimized, each of them in a separate clock domain (CD): two LEON3 SPARC V8 processors [8], a co-processor [8], DMA engine, DRAM controller and shared AMBA bus, as shown in Figure 2.

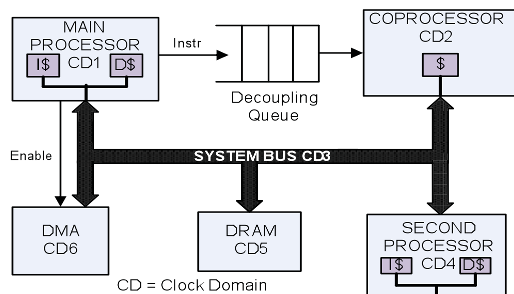


Figure 2: Evaluation system with six clock domains.

A. VHDL-level Instrumentation

We modified the VHDL source code of the LEON3 design to log the transitions of control signals by adding request (*req*) and acknowledge (*ack*) signals between adjacent pipeline stages, without modifying the functionality. When a pipeline stage is ready to send data to the succeeding stage, it asserts the *req* signal (same as the write enable signal of the latch register). The *ack* signal is asserted when the following stage is ready to operate on the data. Overall, we annotated less than 0.2% of the signals in the SoC. Our annotated code increased the system's line count by 1%. Finally, to explore the impact of a large configuration space, we added support for multiple clock domains (CD). Each component, including the bus, is in a separate CD; the frequency of each CD can be adjusted independently.

SoCs contain third party IP blocks for which designers do not always have source code access. We emulate this case by treating the coprocessor and DMA engine as black boxes. For these IP blocks we only use the control signals at the interfaces when computing the GCP, thus reducing instrumentation effort, but potentially sacrificing fidelity.

V. Experimental Evaluation

We perform cycle-accurate behavioral simulation of the design's RTL using ModelSim 6.3 (structural simulation of the system can be used and should produce identical results). Logging all control signals in our system did not increase the simulation time.

SoC designers impose design performance constraints that can be specified by cost functions such as power-delay, area-delay, etc. Cost functions typically include factors such as performance coupled with chip power, area, or other metrics. We define our example cost function as the power-delay product (**PD**), summed over all the components in the SoC:

$$PD = Power \times Delay = \sum(C_i V_i^2 f_i) \times (Exec.\ Time)^1$$

We report normalized power-delay results with respect to the initial configuration. In all of our experiments, we execute a different small, synthetic benchmark on the processors. The main processor executes an integer-heavy computation, while the second processor executes an I/O benchmark. The two processors run concurrently, and compete with each other for resources, such as the shared system bus. The coprocessor inspects the instruction stream committed by the main processor, and checks for security flaws. While our benchmarks are small (hundreds of thousands of cycles), our methodology can be easily extrapolated to more complex workloads.

A. Search Space Exploration

We first performed an exhaustive search of the parameter space for 3 independent parameters: the clock frequencies of the second CPU, the coprocessor, and DRAM. (The clock frequency of the main CPU is constant; frequencies are changed in 5MHz increments). We constrain system performance to be above a minimum threshold; an execution longer

[1] C is the capacitance, V is the voltage and f the frequency of each system component *i*.

than the threshold is unacceptable and not shown in the surfaces in Figure 4. As a result, the search space has an irregular shape. Figure 4 shows the Power-Delay (PD) values for all possible legal combinations, where a high PD is bad, and a low PD is good.

GCP directed Search Algorithm

Step 1: *Select initial configuration parameters for different IP blocks such that cost function is satisfied.*
Step 2: *Perform simulation of workload*
Step 3: *If performance of system is worse than previous performance, STOP, else proceed to step 4*
Step 4: *Using GCP analysis, identify bottlenecks.*
Step 5: *Optimize configuration parameters for the bottleneck IP block i.e. find a pair of IP blocks to "swap". Make the one on the critical path faster, and the other one (which is off the critical path) slower, without exceeding the cost-function.*
Step 6: *Go to Step 2 (iterate).*

Figure 3: Algorithm of the GCP-directed search.

We then performed a directed search on the configuration space, using information provided by the GCP. Figure 3 provides a generic algorithm for a GCP directed search. The search proceeds by choosing one of two kinds of moves: (1) increase system performance by speeding up a module on the critical path, or (2) decrease system power by slowing down a module outside of the critical path. More sophisticated algorithms can be formulated and used in this framework. It is important to note that the GCP merely provides information on the modules that bottleneck system-wide progress, and potential optimization points. The GCP information is used to perform a “swap” on IP blocks: the IP block on the critical path is made faster at the expense of the other block (off the critical path). This swap guarantees that the GCP-directed search progresses monotonically, allowing for quicker convergence. Using the search algorithm in Figure 3 gives us a local minimum as opposed to the global minimum (the optimal solution). The GCP can also be coupled with more sophisticated search techniques such as simulated annealing, to speed up convergence to optimal solutions.

Note that while we only modify clock frequencies of components in these experiments, we could choose other moves that impact the cost function, such as capacitance, voltage, even arbiter priorities and cache sizes.

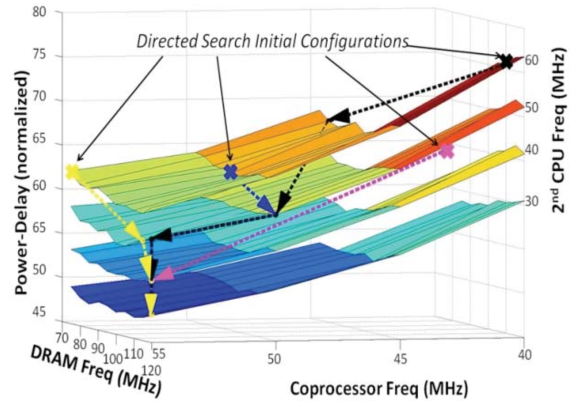
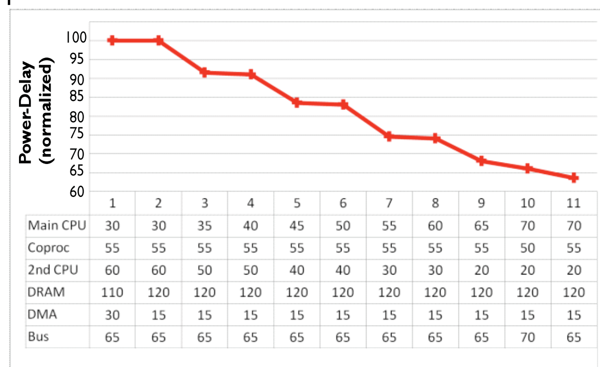


Figure 4: Complete search space for 4-module system when varying the frequency of the 2nd CPU, coprocessor and DRAM. The four surfaces correspond to the four legal values of the 2nd processor’s frequency. The colored arrows show the directed search followed by using the GCP from 4 initial random points.

Computing these results required more than 160 simulations when exploring just three degrees of freedom. The GCP-based directed search uses far fewer simulation points in the search space while improving the optimization criterion, PD, as delineated by the thick arrows for 4 searches (starting from random initial configurations) in Figure 4, which take at most 5 steps. This directed search is completely automatic, and does not require any human intervention.

By making all 6 IP blocks in our system configurable, the size of the search space grows from 160 to 19200, making exhaustive search infeasible. For such a large space, we cannot exhaustively compute the [near] optimal configuration. This issue is even more acute for real systems, which can have tens or hundreds of degrees of freedom. In Figure 5, we show the results of the directed search for the large search space, which converges to a minimal **PD** configuration in just 11 steps. This is the longest run out of the multiple simulations that we performed.



	1	2	3	4	5	6	7	8	9	10	11
Main CPU	30	30	35	40	45	50	55	60	65	70	70
Coproc	55	55	55	55	55	55	55	55	55	50	55
2nd CPU	60	60	50	50	40	40	30	30	20	20	20
DRAM	110	120	120	120	120	120	120	120	120	120	120
DMA	30	15	15	15	15	15	15	15	15	15	15
Bus	65	65	65	65	65	65	65	65	65	70	65

Figure 5: Directed search in a 6-dimensional space.

B. Abstracting module implementations

We treated the CPU as a black box and compared the resulting GCP with that obtained with knowledge of the internal CPU structure. We found that both analyses ranked the same edges in the histogram to be critical. There was a slight difference of 3% in the number of transitions seen between the abstracted and non-abstracted case. On further investigation, we found that this difference was due to the non-blocking stores issued by the main processor that hit in its data cache. In the non-abstracted view, these stores are not considered critical because the processor does not stall. This shows promise for abstracting low-level detail in IP blocks resulting in less logging overhead and closed-source IP block compatibility.

VI. Conclusions

We demonstrated the use of GCP analysis for diagnosing and optimizing performance problems in GALS MPSoC systems where the designer may not understand complex system interactions. Our model MPSoC consisted of GALS components. We showed that a directed search algorithm based on the GCP provides optimal configurations in a few steps (11 out of 19200 possibilities). We successfully applied this technique to SoC designs with 3-6 degrees of freedom.

Our initial implementation required knowledge of the system in order to instrument the source code. We instrumented less than 0.2% of the module signals and added 1% more lines of instrumentation code, introducing negligible overhead to the simulation time. By abstracting RTL modules, the absolute difference of the GCP analysis when compared to complete GCP obtained using the low-level HDL analysis was only 3%. Additionally, the overall GCP ranking of module criticality was unchanged.

In future work, we would like to automatically infer control signals from the HDL, generate the resulting instrumentation code, and generate accurate abstract SoC models, which will speed up simulation for real, commercial MPSoCs or SoCs. Finally, we would like to apply the GCP to larger SoC and MPSoC systems, and explore hardware-based collection techniques using FPGAs.

REFERENCES

[1] P. Barford and M. Crovella, “Critical Path Analysis of TCP transactions”, *Proc. IEEE Transactions on Networking*, 2001

[2] E. Borch, et al., “Loose loops sink Chips”, *Proc. HPCA*, Feb 2002

[3] S. M. Burns. Performance Analysis and Optimization of Asynchronous Circuits. PhD thesis, California Institute of Technology, 1991

[4] J. D. Davis, J. Laudon, and K. Olukotun, “Maximizing CMP Throughput with Mediocre Cores”, *Proc. PACT, Sep 2005*

[5] T. Eeckelaert, et al., “Efficient Multiobjective Synthesis of Analog Circuits using Hierarchical Pareto-Optimal Performance”, *Proc. DATE*, 2005

[6] B. Fields, et al., “Focusing processor policies via critical-path prediction”, *Proc. ISCA,* Jun 2001

[7] F. Ghenassia (ed), “Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems”, Springer, 2005

[8] H. Kannan, M. Dalton, and C. Kozyrakis, “Decoupling Dynamic Information Flow Tracking with a Dedicated Coprocessor**”,** *Proc DSN*, Jun 2009

[9] H. Kannan et al., “Tuning SoCs using the Dynamic Critical Path”, Microsoft Research Tech. Report, MSR-TR-2009-44, Apr 2009

[10] LEON3 SPARC Processor. http://www.gaisler.com.

[11] R. Nagarajan, et al., “Critical Path Analysis of the TRIPS Architecture”, *Proc. ISPASS*, Apr 2006

[12] A. Saidi, et al., “Full-system critical path analysis”, *Proc. ISPASS*, Apr 2008

[13] G. Venkataramani, et al., “Critical Path: A Tool for System-Level Timing Analysis”, *Proc. DAC*, Jun 2007

[14] G. Venkataramani and S. Goldstein, “Operation chaining asynchronous pipelined circuits”, Proc. ICCAD, Nov 2007

[15] A. Xie, et al, “Bounding average time separations of events in stochastic timed Petri nets with choice”, *Proc. ASYNC*, Apr 1999