# Rapid Resource-Constrained Hardware Performance Estimation

Basant K. Dwivedi
Calypto Design Systems (I) Pvt. Ltd.
Som Datt Tower, K-2, Sector 18, NOIDA, India
basant@calypto.com

Arun Kejariwal
School of Information and Computer Science
University of California at Irvine, USA
arun_kejariwal@ieee.org

M. Balakrishnan
Dept. of Computer Science and Engg.
Indian Institute of Technology Delhi
Hauz Khas, New Delhi, India
mbala@cse.iitd.ernet.in

Anshul Kumar
Dept. of Computer Science and Engg.
Indian Institute of Technology Delhi
Hauz Khas, New Delhi, India
anshul@cse.iitd.ernet.in

## Abstract

In a hardware-software co-design environment, an application is partitioned into modules. Each module is then mapped either to software or to hardware. The mapping process is driven by the hardware/software cost and performance parameters of each module. This makes hardware estimation important to evaluate the various candidate architectures. Lack of an efficient hardware estimation methodology and a supporting tool results in inefficient partitioning. In this paper, we present novel algorithms for clock period estimation and estimation of upper bound on execution time under given resource constraints which includes constraints on number of ports in the register file and memory. Experimental results on benchmarks from the High-Level Synthesis (HLS) [1], MiBench [2] and Mediabench [3] suites, show the effectiveness of our algorithms.

## 1 Introduction

With decreasing cost of silicon, system-on-chip (SoC) based architectures are becoming common. SoC architectures for many applications consist of programmable processors which execute software (SW) part of the application and hardware (HW) accelerators [4, 5] which execute the hardware part of the application. Synthesis of such systems is achieved by some hardware-software co-design frameworks such as [6, 7, 8].

A typical HW/SW co-design framework is shown in Figure 1. An application written in a high-level language like C along with an architectural template is fed to the partitioner. The partitioner then maps functions of the application onto HW or SW based on corresponding HW/SW metrics such as area, performance, power etc. SW metrics can be obtained by simulation or actual execution on the target processor, but it takes more time. On the other hand SW estimation such as done in [9] is faster. Similarly HW metrics can be obtained either by doing synthesis or by performing estimation [10, 11]. As shown in this paper, getting HW metrics using estimation is much faster than actual synthesis. Such an estimation approach is necessary for providing values of HW metrics during iterative design space exploration (DSE) process.

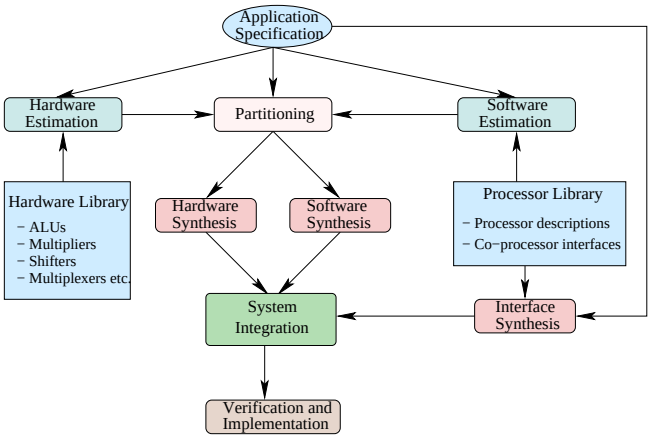


**Figure 1. A HW/SW co-design framework**

In this paper, we address the estimation of performance of the given part of application when it is mapped onto HW. Specifically, performance or execution time is a product of number of clock cycles and clock period. The choice of a particular clock period influences the utilization of functional units used in the design apart from the execution time.

Thus, optimal clock estimation is critical in a co-design environment especially for real time embedded systems because of real-time performance constraints. In [12], maximum delay of a functional unit (FU) in the critical path is chosen as the clock cycle. In [13], Yoda et. al present gate-level *delay-insertion* based clock period minimization of synchronous circuits. However, such low-level tuning is not possible because gate level information is not available. The *Clock Slack Minimization (CSM)* method [14] estimates the "optimum" clock by minimizing the average slack in the entire data flow graph of the program. This method does not consider the critical path which results in longer execution times. In Section 3.1, we present *Critical Path Slack Minimization (CPSM)* algorithm which takes into account the critical path to determine optimal clock.

For performance estimation, the other parameter to be estimated is the number of clock periods. Unlike work on estimation of lower bound [15, 16, 17] on execution time and architectural resources requirements [16, 18, 19], problem of estimation of upper bound on execution time has not been adequately addressed. For applications which are required to meet real-time constraints, upper bound estimation is necessary to ensure a feasible design. In [20], Lin et. al present *Operator-Use Method (OUM)* to estimate the upper bound on the number of control steps required to execute a behavior, given a resource constraint set. The upper bound on control steps for a ready list (refer to Section 3.2) is given by Equation (1)

$$csteps = \max_{\forall \phi \in \Phi} \left( \left\lceil \frac{occur(\phi)}{num(\phi)} \right\rceil \times delay(\phi) \right) \qquad (1)$$

where, $\Phi$ represents the set of all FU types. $occur(\phi)$ and $num(\phi)$ represent the number of occurrences of operations which are to be mapped to FU type $\phi$ in a basic block and number of instances of FU type $\phi$ respectively. $delay(\phi)$ represents the delay of an FU type $\phi$. *OUM* has $O(n)$ complexity where $n$ is the number of nodes in the data flow graph considered. However, they do not take int account the register file and memory port constraints for estimation purposes.

Parallel execution of two or more instructions (like in multiple issue processors) mandates simultaneous access to operands stored in the register file or in the memory. This has made the use of multiport register files and memory indispensable - as they permit reading/writing more data during one clock cycle. Thus, register file and memory port constraints must be considered during hardware performance estimation.

Jie et. al [21] compute the upper bound on execution time using a variant of *list scheduling*. Though they consider port constraint, the complexity of their approach is $O(n^2)$. Also, the scheduling stage will become the bottleneck, if a number of different hardware implementations

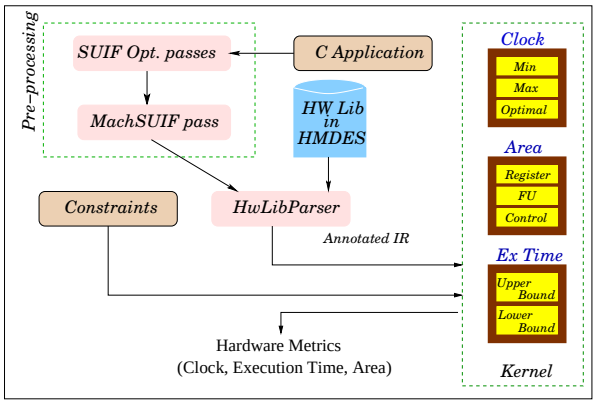


**Figure 2. $\mathcal{HwEst}$: Hardware Estimator Tool**

are to be evaluated corresponding to different resource constraints. Except in [20, 21], estimation of upper bound on performance has not been reported by anyone to the best of our knowledge.

In Section 3.2, we present a novel algorithm to estimate upper bound on execution time. This algorithm takes into account the constraints due to limited number of ports in the register file and memory. The complexity of our algorithm is $O(n)$ which enables to quickly evaluate various hardware implementations for different resource constraints.

The rest of the paper is organized as follows. In Section 2, we discuss our hardware estimation tool setup. Section 3.1 and Section 3.2 provide details of our clock period and upper bound execution time estimation algorithms. Section 4 presents the experimental results and finally, Section 5 summarizes this work and discusses the possible future directions.

## 2 Hardware Estimation Tool

The algorithms presented in this paper have been implemented in our hardware estimator tool, $\mathcal{HwEst}$ shown in Figure 2. An application written in C language is preprocessed to generate the program's SUIF2 [22] IR. Then classical optimizations such as common sub-expression elimination (CSE) available in SUIF are applied on the IR. The MachSUIF [23] passes are used to generate the control flow graph (CFG) of the program and data dependence graph (DDG) at the basic block level. The *HwLibParser* pass then annotates the library information in the IR. We used HMDES [24] to describe our parameterizable resource library. The annotated IR is fed to the $\mathcal{HwEst}$'s kernel along with the constraints set. The kernel consists of independent passes, each estimating a particular metric. The kernel supports clock estimation, area estimation and lower and upper bounds on execution time. Except clock and upper bound execution time estimations, detailed discussion on rest of the stages of $\mathcal{HwEst}$ is outside the scope of this paper.

**Algorithm 1** Critical Path Slack Minimization (CPSM)

/* Determine the critical path */
$critical\_path \leftarrow$ path with maximum $PL(P_i) \times \Gamma(P_i)$ out of all the paths in CDFG
**for** each $\phi \in \Phi$ in the critical_path **do**
  calculate load($\phi$)
**end for**
$Clk_{opt} = 0$
**for** $Clk_{min} \leq Clk \leq Clk_{max}$ **do**
  **for** each $\phi \in \Phi$ **do**
    /* Compute slack */
    $slack(\phi) = \left\lceil \frac{delay(\phi)}{Clk} \right\rceil \times Clk - delay(\phi)$
  **end for**
  /* Compute total slack, $\Omega$, along the critical path */
  $\Omega(Clk) = \sum_{\phi \in \Phi} slack(\phi) \times load(\phi)$
  /* Compute utilization, $\mu$ */
  $\mu = 1 - \frac{\Omega(Clk)}{PL(critical\_path)}$
  **if** $\Omega(Clk)$ = *maxima* **then**
    $Clk_{opt}$ = Clk
  **end if**
**end for**

# 3 Execution Time Estimation Methodology

## 3.1 Clock Estimation

Optimal clock selection is guided by the following objectives - 1) Minimizing execution time and 2) Maximizing clock utilization. In most applications critical path execution delimits overall performance. Therefore, optimal clock is determined w.r.t. the critical path. Optimal clock is determined by minimizing the total slack along the critical path. The clock period thus obtained minimizes execution time of the program and maximizes FU utilization.

We developed Algorithm 1, *Critical Path Slack Minimization (CPSM)*, to determine optimal clock by minimizing clock slack along the critical path (*critical_path*) in the control data flow graph (CDFG) of the application. In Algorithm 1, $PL(P_i)$ and $\Gamma(P_i)$ denote the path length and execution frequency of path $P_i$ respectively. $\Phi$ is a set of FU types and *load($\phi$)* represents the number of instances of an FU type $\phi$ along the critical path. The delay of an FU type $\phi$ is denoted by delay($\phi$). $Clk_{min}$ is governed by the maximum clock frequency specified by design libraries. $Clk_{max}$ is approximated to the smallest value of operator delay. We compute clock utilization. $\mu$, for each $Clk$ where $Clk_{min} \leq Clk \leq Clk_{max}$. The optimal clock corresponds to the maximum clock utilization - *maxima* in Algorithm 1.

**Example 1** *Consider the DDG given in Figure 3. The flow along the bold arrows represents the critical path of the DDG. For this DDG, CSM yields an optimum clock of 74 ns. However, slack minimization along the critical path yields an optimum clock of 30 ns. Table 1 presents quantitative analysis of CPSM and CSM. We observe that CSM results in longer execution times with more slack penalty. CPSM eclipses CSM in clock utilization by over 5%. Experiments on Mediabench, MiBench, HLS benchmarks, Section 4.1, validate the above.*

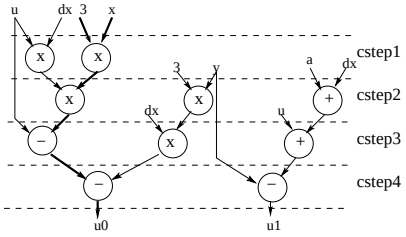


**Figure 3. Optimal Clock Determination**

| | CPSM | CSM |
|---|---|---|
| Total Ex. Time | 420 ns | 444 ns |
| $\Omega$ | 14 ns | 38 ns |
| $\mu$ | 96.67% | 91.44% |

**Table 1. Comparison between CPSM & CSM**

In Figure 3 the critical path corresponds to the longest path of the DDG. However, critical path in a *CDFG* of the application may not correspond to the longest path. For example, consider the CDFG shown in Figure 4. The path lengths (PL) in Figure 4 are $PL(Path_1) = PL(Path_2) = 18$ *cycles*, $PL(Path_3) = PL(Path_4) = 13$. The corresponding execution frequencies are shown in parentheses. Though $PL(Path_1) > PL(Path_3)$, Path 3 is the critical path of the CDFG from Algorithm 1.

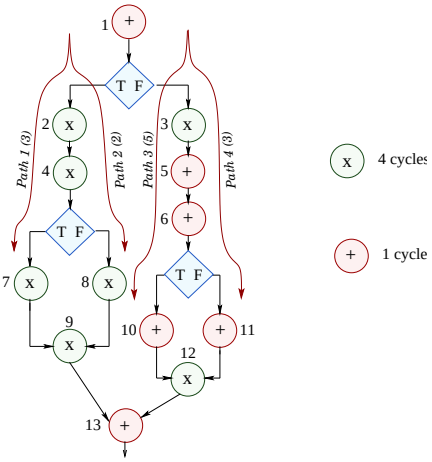


**Figure 4. Critical Path Determination in a CDFG**

*CSM* estimates "optimal" clock for each DDG (an input to the algorithm) of the CDFG. Hence, for each DDG the

clock frequency should be set equal to the local optimum. However, such dynamic tuning is non-trivial and infeasible. On the contrary, Algorithm 1 determines global optimal clock corresponding to the critical path of the CDFG. Our experiments, Section 4.1, show the effectiveness of our approach.

Critical path is determined with a variant of the Dijkstra's algorithm [25]. A Fibonacci heap implementation of the priority queue yields a time complexity $O(E+VlogV)$, where $V$ is the number of vertices and $E$ is the number of edges in the CDFG. The overall complexity of Algorithm 1 is $O((E+VlogV)+kv)$, where $v$ is the number of vertices in the critical path and k is given by

$$k = \left\lceil \frac{Clk_{max} - Clk_{min}}{\Delta} \right\rceil$$

$\Delta$ represents the increment in clock period in the *for* loop in Algorithm 1.

### 3.2 Estimation of Execution Time Upperbound

In our methodology, we model ports as resources like other functional units (FUs). Read and write ports have distinct resource identities. We developed the *Resource Use Method (RUM)* (Algorithm 2) to encapsulate the effect of port contention on execution time.

A *ready_list* (enclosed by dashed lines in Figure 5) is defined as a set of data independent operations in a basic block. Upper bound on execution time of a basic block, $UB(BB_i)$, is a maximum of the multiple upperbounds resulting from individual resource constraints. First, we compute FU constrained upperbound $UB_{FU}$ for a *ready_list*. We account for FUs of both types - pipelined/non-pipelined. Then, we compute port-constrained upperbound of the *ready_list*, denoted by $UB_{Port}$. For the same, we compute *memory-bound*, denoted by $UB_{Port_{Mem}}$, and *register-bound*, denoted by $UB_{Port_{Reg}}$ upper bounds. Read and write port constraints affect the latter independent of each other. Therefore, we compute $RF_{read}$ and $RF_{write}$, which correspond to read and write port requirements for parallel execution of all operations in a ready list respectively. In case of non-pipelined FUs we assume that data needs to be latched for the entire delay of the operation. We encapsulate the multi-cycle port occupancy of non-pipelined FUs by weighing $rf_{rp}(\vartheta)$ with $delay(\vartheta)$, where $\vartheta \in (\Phi - \{ld, st\})$. Note that write port requirement of a MIMO (multiple-input multiple output FU), $rf_{wp}(\vartheta)$, is more than one. Let us walk through an example for better understanding of Algorithm 2.

**Algorithm 2** Resource Use Method

UB($BB_i$) = 0
**for each** ready list in $BB_i$ **do**
  **for each** Op $\phi \in \Phi$ **do**
    /* Compute FU constrained upperbound */
    **if** *pipelined* **then**
      $UB_{FU}(\phi) = \left( \left\lceil \frac{occur(\phi)}{num(\phi)} \right\rceil \times \frac{delay(\phi)}{pipe\_stages(\phi)} \right)$
    **else**
      $UB_{FU}(\phi) = \left( \left\lceil \frac{occur(\phi)}{num(\phi)} \right\rceil \times delay(\phi) \right)$
    **end if**
  **end for**
  $UB_{FU} = \max_{\forall \phi \in \Phi} UB_{FU}(\phi)$
  /* Compute port constrained upperbound */
  $UB_{Port_{mem(read)}} = \left( \left\lceil \frac{occur(ld)}{num(\mathcal{M}_{read})} \right\rceil \times delay(ld) \right)$
  $UB_{Port_{mem(write)}} = \left( \left\lceil \frac{occur(st)}{num(\mathcal{M}_{write})} \right\rceil \times delay(st) \right)$
  $UB_{Port_{mem}} = max\ (UB_{Port_{mem(read)}}, UB_{Port_{mem(write)}})$
  /* Let $\Psi = \Phi - \{ld, st\}$ */
  **if** *pipelined* **then**
    $RF_{read} = \sum_{\vartheta \in \Psi} (rf_{rp}(\vartheta) \times occur(\vartheta))$
  **else**
    $RF_{read} = \sum_{\vartheta \in \Psi} (rf_{rp}(\vartheta) \times occur(\vartheta)) \times delay(\vartheta)$
  **end if**
  $RF_{write} = \sum_{\vartheta \in \Psi} rf_{wp}(\vartheta) \times occur(\vartheta)$
  $UB_{Port_{reg(read)}} = \left\lceil \frac{RF_{read}}{num(\mathcal{RF}_{read})} \right\rceil$
  $UB_{Port_{reg(write)}} = \left\lceil \frac{RF_{write}}{num(\mathcal{RF}_{write})} \right\rceil$
  $UB_{Port_{reg}} = max\ (UB_{Port_{reg(read)}}, UB_{Port_{reg(write)}})$
  $UB_{Port}$ = max ($UB_{Port_{Mem}}, UB_{Port_{Reg}}$)
  UB($BB_i$) += $max\ (UB_{FU}, UB_{Port})$
**end for**

Where:

$num(\mathcal{M}_{read})$ = Number of memory read ports.
$num(\mathcal{M}_{write})$ = Number of memory write ports.
$num(\mathcal{RF}_{read})$ = Number of register file read ports.
$num(\mathcal{RF}_{write})$ = Number of register file write ports.
$pipe\_stages(\phi)$ = Number of pipeline stages in Op($\phi$)
$rf_{rp}(\vartheta)$ = number of read ports required by Op($\vartheta$)

for e.g. **add** r2, r1, r0 $\rightarrow$ $rf_{rp}(add) = 2$

**addi** r2, r1, 2 $\rightarrow$ $rf_{rp}(addi) = 1$

$rf_{wp}(\vartheta)$ = Number of write ports required by Op($\vartheta$).
$occur(\phi)$ = Number of occurrences of $\phi$ in a ready list.

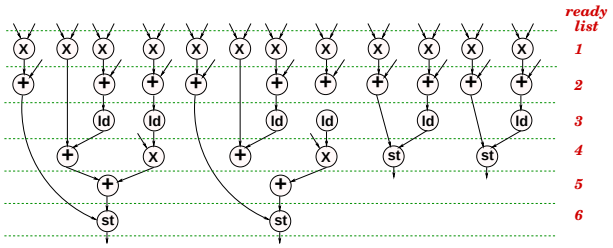


**Figure 5. DDG of the function predict_state**

| Operator | + | × | ld | st | $\mathcal{RF}_{read}$ | $\mathcal{RF}_{write}$ | $\mathcal{M}_{read}$ | $\mathcal{M}_{write}$ |
|---|---|---|---|---|---|---|---|---|
| Delay (cycles) | 4 | 5 | 4 | 4 | 4 | 2 | 2 | 1 |
| # | 4 | 3 | 2 | 2 | | | | |

**Table 2. Hardware Library**

**Example 2** *Consider the DDG (from the predict_state function of Kalman Filter) shown in Figure 5 with resource library given in Table 2.*

*Here we assume non-pipelined units. Let us consider ready list 1. The resource-constrained upperbound is*

$$UB_{FU} = \left\lceil \frac{occur(\times)}{num(\times)} \right\rceil \times delay(\times) = 20 \; cycles$$

*The port-constrained upperbound is determined as follows :*

$$\begin{aligned} RF_{read} &= rf_{rp}(\times) \cdot occur(\times) \cdot delay(\times) \\ &= 2 \cdot 12 \cdot 5 = 120 \\ RF_{write} &= rf_{wp}(\times) \cdot occur(\times) = 1 \cdot 12 = 12 \\ UB_{Port_{reg(read)}} &= \frac{RF_{read}}{num(\mathcal{RF}_{read})} = 30 \\ UB_{Port_{reg(write)}} &= \frac{RF_{write}}{num(\mathcal{RF}_{write})} = 6 \end{aligned}$$

*Thus,*

$$\begin{aligned} UB_{Port} &= UB_{Port_{Reg}} \\ &= max\left(UB_{Port_{reg(read)}}, UB_{Port_{reg(write)}}\right) \\ &= 30 \; cycles \end{aligned}$$

*Note that $UB_{Port} > UB_{FU}$. Thus, $UB = max\,(UB_{Port}, UB_{FU}) = 30$ unlike OUM which reports an upperbound of 20 cycles (= $UB_{FU}$).*

Example 2 highlights the fact that execution time may be bounded due to port contention irrespective of the amount of computing power available.

Complexity of the Algorithm 2 is bounded by the loop which computes $UB_{FU}$ as this loop is repeated for each operation type. Generation of all the ready lists of DFG does not take more than $O(n)$ time, so the overall complexity of the algorithms is $O(mn)$, where $n$ is the number of nodes (operations) in the DFG and $m$ is the number of operation types. In practice $m$ is small, hence effective time complexity of Algorithm 2 is $O(n)$.

# 4 Results

In this section, we present experimental results for optimal clock selection and our exploration results for the effect of port constraints on execution time. The resource library for the estimation of upper bound was specified in a manner so as to support maximum concurrency. The upperbound estimates presented in Sections 4.2, 4.3 correspond to the entire application and were obtained by a weighted summation of the upperbounds of individual basic blocks. The weight of each basic block corresponds to it's execution frequency which in turn was obtained by profiling the application. We validate our estimates against synthesis results in Section 4.4.

## 4.1 Optimal Clock Estimation

In Table 3, we present quantitative analysis of *CPSM* and *CSM*. The critical path execution time was used as a measure of "quality" of the clock estimate. Clock utilization ($\mu$) is determined by calculating the total clock slack, $\Omega$, along the critical path. From Table 3 we observe that $Clk_{opt}$ obtained from *CPSM* yields better clock utilization. We note that in most cases the optimal clock period is guided by the frequency of different operation types i.e. an application with "heavy" load(+) has $Clk_{Opt} \approx delay(+)$. However, in cases like *Pipelined FIR Filter*, where the number of different operation types is evenly distributed, the optimal clock period does not correspond to the delay of any FU.

We also analyzed the effect of clock selection on static power consumption. For the same, we synthesized the applications for the two different clock frequencies - $Clk_{opt_{CPSM}}$ and $Clk_{opt_{CSM}}$. As expected, at higher frequency - $Clk_{opt_{CPSM}}$ - clock utilization increases, but power consumption also increases[1]. So, clock selection should be made in conjunction with the associated performance-power tradeoff.

Thus, *CPSM* scores over *CSM* in both - minimizing execution time and maximizing clock utilization. Therefore, *CPSM*-based optimal clock selection is imperative from performance point of view.

## 4.2 Effect of Register File Port Constraint

Table 4 lists the estimates of upperbound on execution time for different register file port configurations. Each column corresponds to a different port configuration, given by the tuple – <# of read ports, #of write ports >. We observe a decrease in upperbound estimates with increase in number of ports. This can be attributed to the fact that multiple ports support parallel read/write, thus enhancing parallel execution. Note that the upper bound corresponding to

[1] Note that power consumption is a function of both - critical path length and clock period.

| | | CPSM | CSM | | | CPSM | CSM | | | CPSM | CSM |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ADPCM Coder | $Opt_{clk}$ (ns) | 49 | 56 | ADPCM Decoder | $Opt_{clk}$ (ns) | 48 | 56 | Elliptic Wave Filter | $Opt_{clk}$ (ns) | 16 | 49 |
| | $\mu$ | 93.19 | 90.69 | | $\mu$ | 95.87 | 91.69 | | $\mu$ | 94.71 | 91.37 |
| | Power ($\mu$W) | 146.7 | 56.44 | | Power ($\mu$W) | 124.8 | 48.3 | | Power ($\mu$W) | 48.3 | 18.5 |
| Pipelined FIR Filter | $Opt_{clk}$ (ns) | 14 | 56 | Wavelet | $Opt_{clk}$ (ns) | 48 | 74 | Kalman Filter | $Opt_{clk}$ (ns) | 56 | 74 |
| | $\mu$ | 97.51 | 96.86 | | $\mu$ | 93.09 | 86.73 | | $\mu$ | 96.2 | 91.33 |
| | Power ($\mu$W) | 35.60 | 21.43 | | Power ($\mu$W) | 41.17 | 29.60 | | Power ($\mu$W) | 79.6 | 41.2 |
| Differential Equation | $Opt_{clk}$ (ns) | 19 | 74 | FFT | $Opt_{clk}$ (ns) | 49 | 74 | Cubic | $Opt_{clk}$ (ns) | 48 | 56 |
| | $\mu$ | 97.13 | 91.44 | | $\mu$ | 96.30 | 93.55 | | $\mu$ | 95.89 | 91.23 |
| | Power ($\mu$W) | 38.4 | 22.5 | | Power ($\mu$W) | 39.8 | 24.4 | | Power ($\mu$W) | 54.4 | 32.4 |

**Table 3. Optimal Clock Estimation**

| | **RUM** (cycles) | | | | | | | | **OUM** (cycles) |
|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | ($\mathcal{RF}_{read}$,$\mathcal{RF}_{write}$) | | | | ($\mathcal{M}_{read}$,$\mathcal{M}_{write}$) | | | | |
| | (2,1) | (4,2) | (4,4) | (6,5) | (2,1) | (4,2) | (4,4) | (6,5) | |
| ADPCM Coder | 1.85e+7 | 1.35e+7 | 1.31e+7 | 1.29e+7 | 1.45e+7 | 1.36e+7 | 1.29e+7 | 1.29e+7 | 1.29e+7 |
| ADPCM Decoder | 1.67e+7 | 1.20e+7 | 1.11e+7 | 1.08e+7 | 1.23e+7 | 1.13e+7 | 1.08e+7 | 1.08e+7 | 1.08e+7 |
| Elliptic Wave Filter | 270 | 146 | 110 | 98 | 244 | 146 | 98 | 98 | 98 |
| Pipelined FIR Filter | 480 | 286 | 164 | 164 | 380 | 272 | 164 | 164 | 164 |
| Wavelet | 2320 | 1180 | 580 | 580 | 1020 | 860 | 648 | 580 | 580 |
| Kalman Filter | 3680 | 1920 | 1010 | 768 | 912 | 840 | 768 | 768 | 768 |
| Differential Equation | 90 | 52 | 44 | 44 | 76 | 64 | 44 | 44 | 44 |
| FFT | 536 | 312 | 264 | 212 | 286 | 244 | 212 | 212 | 212 |
| Cubic | 4430 | 2460 | 1446 | 970 | 1284 | 1196 | 1120 | 1120 | 1120 |

**Table 4. Effect of ($\mathcal{RF}_{read}$,$\mathcal{RF}_{write}$) and ($\mathcal{M}_{read}$,$\mathcal{M}_{write}$) on Upper Bound**

the configuration $(6,5)$ is not the lower bound on execution time. Instead, it reflects the parallelism embedded in the application (compare it with the upper bound estimate for the configuration $(2,1)$). The lower bound on execution time can be found using [15, 16, 17] or by scheduling techniques [26].

Our experimental results reflect the amenability of the *Resource-Use Method (RUM)* algorithm to different port configurations. In contrast, we note that the *Operator-Use Method (OUM)* is insensitive to varying number of read/write ports. It reports upperbound on execution time corresponding to maximum parallelism case, which is not applicable in most cases.

From Table 4, we observe that for most benchmarks there is minimal gain in performance with increase in number of register file ports from $(4,4) \rightarrow (6,5)$. Thus, the upperbound estimate of execution time for different port configurations can be used to assess application level parallelism. Such estimates aid in making early design decisions in the design of application specific SoCs.

## 4.3 Effect of Memory Port Constraint

Table 4 also lists the estimates of upperbound on execution time for different memory port configurations. Each column corresponds to a different port configuration, given by the tuple – <# of read ports, #of write ports >. Unlike Section 4.2, we note that *degree* of parallel execution (largely influenced by parallel memory read/write) determines the execution time upperbounds. From Table 4 we note that there is no gain in performance (except Wavelet) with increase in number of memory ports from $(4,4) \rightarrow (6,5)$.

It is interesting to note that the impact of parallel memory read/write is less prominent than register file read/write. This highlights the need for application analysis to minimize the trade-off associated with port assignment to register file and memory.

## 4.4 Validation

We developed synthesizable RTL descriptions (in VHDL) of the benchmarks taken from Mediabench [3], Mibench [2] suite. We used Synopsys Design Compiler (DC) [27] to synthesize the HDL descriptions using LSI 10K technology libraries to obtain metrics like critical path length, and chip area. We validated our estimates with the synthesized values. Table 5 presents a quantitative comparison between the $\mathcal{HWEST}$'s upperbound estimates and synthesized values of the kernel of each benchmark. In these experiments, we assume 4 read and 4 write ports in register file and memory.

From Table 5, we see that our bounds are 20-30% off from the synthesized values. However, such deviation is expected because it is an upperbound and also generated at much higher level of abstraction. In spite of that, its utility in pruning the design space is invaluable. Moreover, the lower run time (by over two orders of magnitude) per exploration cycle makes $\mathcal{HWEST}$-based performance estimation of candidate architectures ideal for rapid DSE.

| Benchmark | $\mathcal{HwEst}$ (cycles) | $\mathcal{HwEst}$ (ns) | DC (ns) | % error | Runtime (s) $\mathcal{HwEst}$ | Runtime (s) DC |
|---|---|---|---|---|---|---|
| ADPCM coder | 47 | 235 | 200.64 | 17.13 | 5.17 | 636.1 |
| ADPCM decoder | 29 | 145 | 120.69 | 16.77 | 4.25 | 390.4 |
| Elliptic Wave Filter | 25 | 125 | 94.85 | 24.12 | 3.71 | 380.1 |
| Pipelined FIR Filter | 27 | 135 | 95.67 | 29.13 | 3.73 | 370.6 |
| Wavelet | 37 | 185 | 145.41 | 21.4 | 4.41 | 577.9 |
| Kalman Filter | 18 | 90 | 73.5 | 22.69 | 3.61 | 373.1 |
| Differential Equation | 31 | 155 | 115.12 | 25.73 | 4.25 | 420.3 |
| FFT | 33 | 165 | 140.86 | 22.69 | 4.3 | 456.3 |
| Cubic | 17 | 85 | 63.75 | 34.8 | 3.29 | 390.4 |

**Table 5. Validation of Upperbound Estimates**

# 5 Conclusion

We presented improved algorithms for optimal clock estimation and estimation of upperbound on execution time. Number of memory and register file ports have also been considered for estimation of execution time. This algorithm can be used to quickly evaluate a very large design space created by variety of resource allocations. Our experimental results show that our clock estimation algorithm improves clock utilization and there is an acceptable correspondence between estimation and synthesis results for upperbound on execution time.

Our future work will focus on communication cost estimation and power consumption estimation along with its integration to $\mathcal{HwEst}$.

# References


[1] ftp://ftp.ics.uci.edu/pub/hlsynth/HLSynth92/.

[2] Mibench. http://www.eecs.umich.edu/mibench/.

[3] Mediabench. newblock http://cares.icsl.ucla.edu/MediaBench/.

[4] S. Note, W. Geurts, F. Catthoor, and H. De Man. Cathedral-III: Architecture-driven high-level synthesis for high throughput DSP applications. In *Proc. of the 28th Design Automation Conf.*, pages 597–602, 1991.

[5] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1355–1371, November 2001.

[6] J. Staunstrup and W. Wolf. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[7] Ptolemy: Heterogeneous Modeling, Simulation, and Design of Concurrent Systems. http://ptolemy.eecs.berkeley.edu/.

[8] J.-M. Chang and M. Pedram. Codex-dp: Co-design of Communicating Systems Using Dynamic Programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):732–744, July 2000.

[9] M. K. Jain, M. Balakrishnan, and A. Kumar. An efficient technique for exploring register file size in ASIP synthesis. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES'02)*, pages 252–261, October, 2002.

[10] E. Macii, M. Pedram, and F. Somenzi. High-level Power Modeling, Estimation, and Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1061–1079, November 1998.

[11] C. Jego, E. Casseau, and E. Martin. Interconnect cost control during high-level synthesis. In *Proceedings of Design Circuits & Integrated Systems Conference*, pages 507–512, November 2000.

[12] A. C. Parker, T. Pizzaro, and M. Mlinar. MAHA : A program for datapath synthesis. In *Proc. of the 23rd Design Automation Conf.*, pages 461–466, July 1986.

[13] T. Yoda and A. Takahashi. Clock Period Minimization of Semi-Synchronous circuits by Gate-Level Delay Insertion. *IEICE Transactions Fundamentals*, E82-A(11):2383–2389, November 1999.

[14] En-Shou Chang, D. Gajski, and S. Narayanan. An optimal clock period selection method based on slack minimization criteria. *ACM Transactions on Design Automation of Electronic Systems*, 1(3):352–370, July 1996.

[15] G. Tiruvuri and M. Chung. Estimation of Lower Bounds in Scheduling Algorithms for High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 3(2):162–180, April 1998.

[16] S. Y. Ohm, F. J. Kurdahi, and N. Dutt. A Unified Lower Bound Estimation Technique for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(5):458–472, May 1997.

[17] M. Narasimhan and J. Ramanujam. On lower bounds for scheduling problems in high-level synthesis. In *Proc. of the 37st Design Automation Conf.*, pages 546–551, June 2000.

[18] S. Chaudhuri and R. Walker. Computing lower bounds on functional units before scheduling. *IEEE Transactions on Very Large Scale Integration Systems*, 4(2):273–279, June 1996.

[19] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE Transactions on Very Large Scale Integration Systems*, 2(1):175–190, June 1993.

[20] N. Dutt, D. Gajski, A. Wu, and S. Lin. *High Level Synthesis : Introduction to Chip and System Design*. Kluwer Academic publishers, 1992.

[21] J. Gong, Daniel D. Gajski, and A. Nicolau. A Performance Evaluator for Parameterized ASIC Architectures. In *European Design Automation Conference*, pages 66–71, 1994.

[22] http://suif.stanford.edu/suif/suif2/index.html.

[23] http://www.eecs.harvard.edu/hube/research/machsuif.html.

[24] The MDES user manual. http://www.trimaran.org/.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, second edition, 2001.

[26] S. Novack and A. Nicolau. Trailblazing: A hierarchical approach to percolation scheduling. In *Proc. Intl. Conf. on Parallel Processing*, pages 120–124, 1993.

[27] Synopsys. http://www.synopsys.com.